Informix Product Family
Informix Global Language Support
Version 6.00

IBM Informix GLS User's Guide

SC27-4499-01

Informix Product Family
Informix Global Language Support
Version 6.00

IBM Informix GLS User's Guide

SC27-4499-01

Note
Before using this information and the product it supports, read the information in "Notices" on page C-1.

Edition

This edition replaces SC27-4499-00.

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this publication should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corporation 1996, 2013.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction vii
About this publication vii
Types of users vii
Software compatibility vii
Assumptions about your locale vii
Demonstration databases viii
What's new in GLS, Version 6.00 viii
Example code conventions ix
Additional documentation ix
Compliance with industry standards x
Syntax diagrams x
How to read a command-line syntax diagram xi
Keywords and punctuation xii
Identifiers and names xii
How to provide documentation feedback xiii

Chapter 1. GLS fundamentals 1-1
Character-representation conventions 1-1
Single-byte characters 1-1
Multibyte characters 1-1
Single-byte and multibyte characters in the same string 1-2
White space characters in strings 1-2
Trailing white space characters 1-2
The GLS feature 1-3
GLS support by IBM Informix products 1-5
A GLS locale 1-9
Code sets for character data 1-9
Character classes of the code set 1-10
Collation order for character data 1-11
End-user formats 1-13
Set a GLS locale 1-16
Locales in the client/server environment 1-16
The default locale 1-21
Set a nondefault locale 1-23
GLS locales with IBM Informix products 1-23
Support for non-ASCII characters 1-24
Establish a database connection 1-24
Perform code-set conversion 1-29
Locate message files 1-32
Customize end-user formats 1-32
Customize date and time end-user formats 1-32
Customize monetary values 1-34

Chapter 2. GLS environment variables 2-1
Set and retrieve environment variables 2-1
GLS-related environment variables 2-1
The CC8BITLEVEL environment variable 2-2
The CLIENT_LOCALE environment variable 2-2
The DBDATE environment variable 2-3
The DBLANG environment variable 2-3
The DB_LOCALE environment variable 2-4
The DBMONEY environment variable 2-5
The DBTIME environment variable (ESQL/C) 2-6
The ESQLMF environment variable 2-7
The GLS8BITFSYS environment variable 2-7

© Copyright IBM Corp. 1996, 2013

The GL_DATE environment variable . . . 2-10
The GL_DATETIME environment variable . . . 2-15
The USE_DTENV environment variable . . . 2-20
The GL_USEGLU environment variable . . . 2-20
The SERVER_LOCALE environment variable . . . 2-21

Chapter 3. SQL features . . . 3-1
Name database objects . . . 3-1
Rules for identifiers . . . 3-1
Non-ASCII characters in identifiers . . . 3-1
Valid characters in identifiers . . . 3-5
Character data types . . . 3-6
Localized collation of character data . . . 3-6
Other character data types . . . 3-10
Handle character data . . . 3-12
Specify quoted strings . . . 3-12
Specify comments . . . 3-13
Specify column substrings . . . 3-13
Specify arguments to the TRIM function . . . 3-17
Search functions that are not case-sensitive . . . 3-18
Collate character data . . . 3-18
SQL length functions . . . 3-27
Locale-sensitive data types . . . 3-31
Handle the MONEY data type . . . 3-31
Handle extended data types . . . 3-33
Handle smart large objects . . . 3-33
Data manipulation statements . . . 3-34
Specify conditions in the WHERE clause . . . 3-34
Specify era-based dates . . . 3-34
Load and unload data . . . 3-35
Data definition statements . . . 3-36
Automatic resizing of the expansion factor . . . 3-38

Chapter 4. Database server features . . . 4-1
GLS support by IBM Informix database servers . . . 4-1
Database server code-set conversion . . . 4-2
Data that the database server converts . . . 4-2
Locale-specific support for utilities . . . 4-3
Non-ASCII characters in database server utilities . . . 4-3
Non-ASCII characters in SQL utilities . . . 4-4
Locale support for C User-defined routines (Informix and DB API) . . . 4-5
Current processing locale for UDRs . . . 4-5
Non-ASCII characters in source code . . . 4-5
Copy character data . . . 4-6
The IBM Informix GLS library . . . 4-7
Code-set conversion and the DataBlade API . . . 4-8
Locale-specific data formatting . . . 4-9
Globalized exception messages . . . 4-9
Globalized tracing messages . . . 4-13
Locale-sensitive data in an opaque data type . . . 4-16

Chapter 5. General SQL API features (ESQL/C) . . . 5-1
Support for GLS in IBM Informix client applications . . . 5-1
Client application code-set conversion . . . 5-1
Globalize client applications . . . 5-3
Globalization . . . 5-3
Localization . . . 5-4
Handle locale-specific data . . . 5-6
Process characters . . . 5-6
Format data . . . 5-6

Avoid partial characters 5-7

Chapter 6. IBM Informix ESQL/C features 6-1
Handle non-ASCII characters 6-1
Non-ASCII characters in host variables 6-2
Generate non-ASCII file names 6-3
Non-ASCII characters in ESQL/C source files. 6-3
Define variables for locale-sensitive data 6-6
Enhanced ESQL/C library functions 6-7
DATE-format functions 6-7
DATETIME-format functions 6-9
Numeric-format functions 6-11
String functions 6-14
GLS-specific error messages 6-14
Handle code-set conversion 6-14
Writing TEXT values. 6-15
The DESCRIBE statement 6-16
The TRIM function 6-17

Appendix A. Manage GLS files A-1
Access GLS files A-1
GLS locale files. A-2
Locale categories A-2
Location of locale files A-6
Other GLS files. A-8
Code-set-conversion files A-8
Code-set files A-10
The IBM Informix registry file (Windows) A-10
Remove unused files A-11
Remove locale and code-set-conversion files. A-11
Remove code-set files A-12
The glfiles utility (UNIX) A-12
List code-set-conversion files A-13
List GLS locale files A-13
List character-mapping files A-14

Appendix B. Accessibility B-1
Accessibility features for IBM Informix products. B-1
Accessibility features B-1
Keyboard navigation B-1
Related accessibility information B-1
IBM and accessibility B-1
Dotted decimal syntax diagrams B-1

Notices C-1
Privacy policy considerations C-3
Trademarks C-3

Index X-1

Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About this publication

This publication describes the Global Language Support (GLS) feature available in IBM® Informix® products.

The GLS feature allows IBM Informix application-programming interfaces (APIs) and IBM Informix database servers to handle different languages, cultural conventions, and code sets. This publication describes only the language-related topics that are unique to GLS.

This publication provides GLS information about IBM Informix database servers for both Microsoft Windows and UNIX.

Also see the *IBM Informix GLS API Programmer's Guide*, a companion document that describes the global language support (GLS) application programming interface (API) available in IBM Informix ESQL/C and IBM Informix DataBlade® modules.

Types of users

This publication is written for system administrators and application developers who want to use the GLS environment to create globalized database management applications with IBM Informix products.

This publication is primarily intended for those users who must use IBM Informix products with a nondefault locale. It assumes that you are familiar with IBM Informix database servers and associated products.

If you need more information about features of your operating system to support non-ASCII characters in file names, path names, and other contexts, see your operating system documentation.

Software compatibility

For information about software compatibility, see the IBM Informix GLS release notes.

Assumptions about your locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation and representation of numeric data, currency, date, and time that is used by a language within a given territory and encoding is brought together in a single environment, called a Global Language Support (GLS) locale.

The IBM Informix OLE DB Provider follows the ISO string formats for date, time, and money, as defined by the Microsoft OLE DB standards. You can override that default by setting an Informix environment variable or registry entry, such as **DBDATE**.

© Copyright IBM Corp. 1996, 2013

If you use Simple Network Management Protocol (SNMP) in your Informix environment, note that the protocols (SNMPv1 and SNMPv2) recognize only English code sets. For more information, see the topic about GLS and SNMP in the *IBM Informix SNMP Subagent Guide*.

The examples in this publication are written with the assumption that you are using one of these locales: en_us.8859-1 (ISO 8859-1) on UNIX platforms or en_us.1252 (Microsoft 1252) in Windows environments. These locales support U.S. English format conventions for displaying and entering date, time, number, and currency values. They also support the ISO 8859-1 code set (on UNIX and Linux) or the Microsoft 1252 code set (on Windows), which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

You can specify another locale if you plan to use characters from other locales in your data or your SQL identifiers, or if you want to conform to other collation rules for character data.

For instructions about how to specify locales, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

Demonstration databases

The DB-Access utility, which is provided with your IBM Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix publications are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases are in the `$INFORMIXDIR/bin` directory on UNIX platforms and in the `%INFORMIXDIR%\bin` directory in Windows environments.

What's new in GLS, Version 6.00

This publication includes information about new features and changes in existing functionality.

For a complete list of what's new in this release, go to http://pic.dhe.ibm.com/infocenter/informix/v121/topic/com.ibm.po.doc/new_features_ce.htm.

Table 1. What's new in IBM Informix GLS User's Guide for version 6.00.xC2

Overview	Reference
Defining separators for fractional seconds in date-time values Now you can control which separator to use in the character-string representation of fractional seconds. To define a separator between seconds and fractional seconds, you must include a literal character between the %S and %F directives when you set the **GL_DATETIME** or **DBTIME** environment variable, or when you call the TO_CHAR function. By default, a separator is not used between seconds and fractional seconds. Previously, the ASCII 46 character, a period (.), was inserted before the fractional seconds, regardless of whether the formatting string included an explicit separator for the two fields.	"The GL_DATETIME environment variable" on page 2-15

Example code conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept that is being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access Informix technical information such as information centers, technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

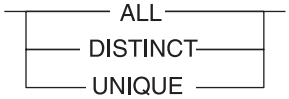
IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

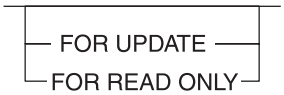
Syntax diagrams

Syntax diagrams use special components to describe the syntax for statements and commands.

Table 2. Syntax Diagram Components

Component represented in PDF	Component represented in HTML	Meaning
	`>>----------`	Statement begins.
	`---------->`	Statement continues on next line.
	`>----------`	Statement continues from previous line.
	`----------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+-----------------+---` `  '------LOCAL------'`	Optional item.

	`---+-----ALL-------+---`
	`   +--DISTINCT-----+`
	`   '---UNIQUE------'`
		Required item with choice. Only one item must be present.

	`---+------------------+---`
	`    +--FOR UPDATE-----+`
	`    '--FOR READ ONLY--'`
		Optional items with choice are shown below the main line, one of which you might specify.

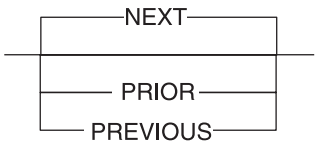
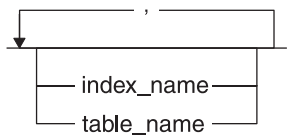
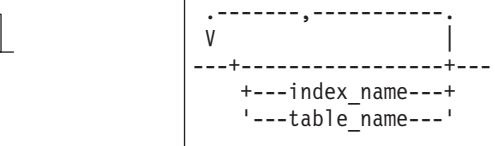
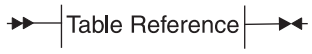
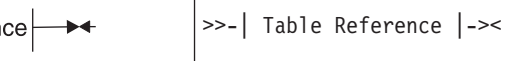
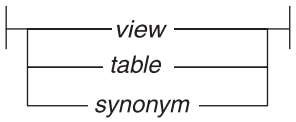
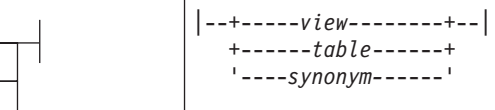
	`    .---NEXT---------.`
	`----+----------------+---`
	`    +---PRIOR--------+`
	`    '---PREVIOUS-----'`
		The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line is used by default.

Table 2. Syntax Diagram Components (continued)

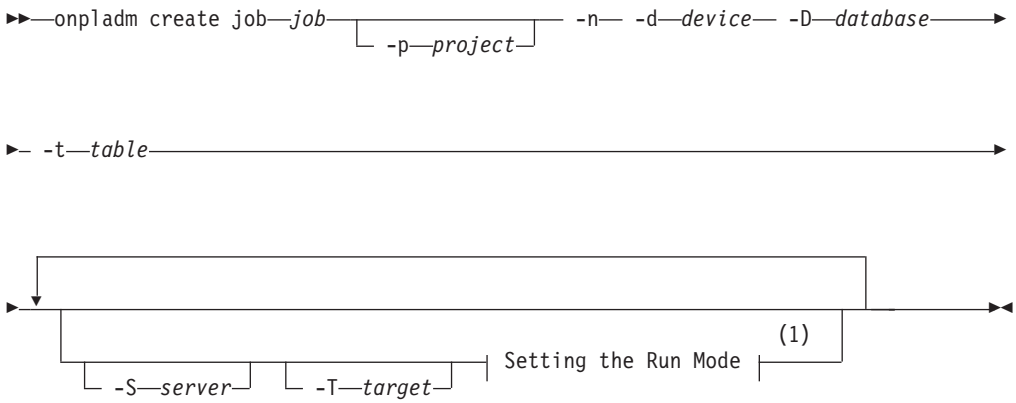
Component represented in PDF	Component represented in HTML	Meaning
		Optional items. Several items are allowed; a comma must precede each repetition.
		Reference to a syntax segment.
Table Reference 	Table Reference 	Syntax segment.

How to read a command-line syntax diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a no-conversion job

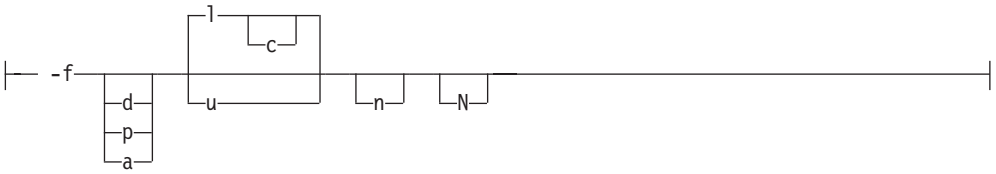


Notes:

1 See page Z-1

This diagram has a segment that is named “Setting the Run Mode,” which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you would find this segment on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the run mode:



To see how to construct a command correctly, start at the upper left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case-sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case-sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Include **onpladm create job** and then the name of the job.
2. Optionally, include **-p** and then the name of the project.
3. Include the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can include one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to include **-f**, optionally include **d**, **p**, or **a**, and then optionally include **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and punctuation

Keywords are words that are reserved for statements and all commands except system-level commands.

A keyword in a syntax diagram is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

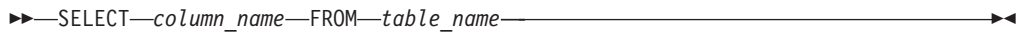
You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in other syntax diagrams. A variable in a syntax diagram, an example, or text, is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.



When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to provide documentation feedback

You are encouraged to send your comments about IBM Informix user documentation.

Use one of the following methods:

- Send email to docinf@us.ibm.com.
- In the Informix information center, which is available online at http://www.ibm.com/software/data/sw-library/, open the topic that you want to comment on. Click the feedback link at the bottom of the page, complete the form, and submit your feedback.
- Add comments to topics directly in the information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. GLS fundamentals

The Global Language Support (GLS) feature lets IBM Informix products handle different languages, cultural conventions, and code sets for Asian, African, European, Latin American, and Middle Eastern countries.

The GLS feature lets you create databases by using the diacritics, collating sequence, and monetary and time conventions of the language that you select. No ONCONFIG configuration parameters exist for GLS, but you must set the appropriate environment variables.

This section introduces basic concepts and describes the GLS feature.

Character-representation conventions

The examples in this documentation use ASCII characters to represent both single-byte and multibyte characters. Multibyte characters are usually ideographic (such as Japanese or Chinese characters). Multibyte and single-byte characters are represented abstractly.

Single-byte characters

Single-byte characters are represented as a series of lowercase letters.

The format for representing one single-byte character abstractly is `a`. Here `a` stands for any single-byte character, not for the letter "a" itself.

The format for representing a string of single-byte characters is `a...z`. Here `a` stands for the first character and `z` stands for the last character in the string. For example, if the string `Ludwig` consists of six single-byte characters, the following format represents this six-character string abstractly:

```
abcdef
```

The letter "s" does not show in examples that represent strings of single-byte characters. The letter "s" is reserved as a symbol to represent a single-byte white space character.

Related concepts:

"White space characters in strings" on page 1-2

Multibyte characters

The appearance of multibyte characters in text, examples, or diagrams are represented abstractly.

The following convention shows abstractly how multibyte characters are stored:

$A^1...A^n$

One to four identical uppercase letters, each followed by a different superscript number, represent one multibyte character. The superscripts show the first to the `n`th byte of the multibyte character, where `n` has values 2 - 4. For example, the following symbols represent a multibyte character that consists of 2 bytes:

A^1A^2

© Copyright IBM Corp. 1996, 2013

The following notation represents a multibyte character that consists of 4 bytes (the maximum length of a multibyte character):

$A^1A^2A^3A^4$

The next example shows a string of multibyte characters in an SQL statement:

CREATE DATABASE $A^1A^2B^1B^2C^1C^2D^1D^2E^1E^2$;

This statement creates a database whose name consists of five multibyte characters, each of which is 2 bytes long.

Related reference:

"Name database objects" on page 3-1

Single-byte and multibyte characters in the same string

For a multibyte code set, a string might be composed of both single-byte and multibyte characters.

To represent mixed strings, this publication combines the formats for multibyte and single-byte characters. The next example represents a string with four characters, where the first and fourth characters are single-byte characters, and the second and third characters are multibyte characters that consist of 2 bytes each:

$aA^1A^2B^1B^2b$

White space characters in strings

White space is a series of one or more characters that show as blank space, Each GLS locale defines what characters are white space characters.

For example, both the TAB (ASCII 9) and blank space (ASCII 32) might be defined as white space characters in one locale, but certain combinations of the **CTRL** key and another character might be defined as white space characters in a different locale.

The convention for representing a single-byte white space in this publication is the letter "s". The following notation represents one single-byte white space:

s

In the ASCII code set, an example of a single-byte white space is the blank character (ASCII 32). To represent a string that consists of two ASCII blank characters, the publication uses the following notation:

ss

The following notation represents a multibyte white space character:

$s^1...s^n$

Here s^1 represents the first byte of the white space character, and s^n represents the last byte of the white space character, where n can range 2 - 4. The following notation represents one 4-byte white space character:

$s^1s^2s^3s^4$

Trailing white space characters

Combinations of characters with white space can occur in quoted strings, in CHAR columns that contain fewer characters than the declared column length, and in other contexts.

For example, if a CHAR(5) column in a single-byte code set contains three characters, the string is padded with two white spaces so that its length is equal to the column length:

```
abcss
```

The next example represents a string of five characters (three characters of data and two trailing white space characters) in a multibyte code set where each of the data characters and white space characters consists of 2 bytes:

$A^1A^2B^1B^2C^1C^2s^1s^2s^1s^2$

In some locales, a string can contain both single-byte and multibyte white space characters. For example, consider the following string:

$abcss^1s^2sss^1s^2$

The string has three single-byte characters (abc), a single-byte white space character (s), a multibyte white space character (s^1s^2), two single-byte white space characters (ss), and one multibyte white space character (s^1s^2).

The GLS feature

In a database application, some of the tasks that the database server and the client application perform depend on the language and culture conventions of the data that they handle.

For example, the database server must sort U.S. English data differently from Korean character data. The client application must show Canadian currency differently from Thai currency.

If the IBM Informix database server or client product included the code to perform these data-dependent tasks, each would need to be written specially to handle a different set of culture-specific data.

With support for GLS, IBM Informix products no longer need to specify how to process culture-specific information directly. Culture-specific information is in a GLS locale. When an IBM Informix product needs culture-specific information, it calls the GLS library, which accesses the GLS locale and returns the information to the IBM Informix product.

The GLS feature is a portable way to support culture-specific information. Although many operating systems provide support for non-English data, this support is usually in a form that is specific to the operating system. Not many standards yet exist for the format of culture-specific information. This lack of conformity means that if you move an application from one operating-system environment to another, you might need to change the way in which the application requests language support from the operating system. You might even find that the new operating-system environment does not provide the same aspect of language support that the initial environment provided.

The GLS feature can access culture-specific information about a UNIX or Windows operating system. IBM Informix products can locate the locale information about any platform to which they are ported.

In order for GLS to support a nondefault locale, the version of Windows that you are using must also support that locale. That is, you cannot support a Japanese client application on Windows unless that application is running on the Japanese version of Windows.

To use the GLS feature, the tasks that you must perform depend on whether you are a system administrator, database administrator, user of a client application, user of a database server utility, or client application developer. The following table lists these optional and mandatory tasks.

Audience	Optional tasks	Mandatory tasks
System administrator, database administrator, or user of client application	• For non-default locales, set the **DB_LOCALE**, **CLIENT_LOCALE**, and **SERVER_LOCALE** environment variables. • To customize end-user formats, set the **GL_DATE**, **GL_DATETIME**, and **DBMONEY** environment variables. For Informix ESQL/C, you can set **DBTIME** instead of **GL_DATETIME**.	None
	• To configure a GLS environment for Informix ESQL/C, set the **CC8BITLEVEL** and **ESQLMF** environment variables. • To perform additional configuration for the GLS environment, set the **DBLANG** and **GLS8BITFSYS** environment variables. • To issue an SQL statement, follow the guidelines in Chapter 3, "SQL features," on page 3-1, and Chapter 4, "Database server features," on page 4-1. • To remove GLS files, follow the guidelines in "Remove unused files" on page A-11. • To get information about GLS files on UNIX, follow the guidelines in "The glfiles utility (UNIX)" on page A-12.	
User of database server utility	Same as above	Follow the guidelines in "Locale-specific support for utilities" on page 4-3.
Client application developer	• Same as above • To develop a globalized client application, follow the guidelines in "Globalize client applications" on page 5-3 and the *IBM Informix GLS User's Guide*.	• Follow the guidelines in Chapter 5, "General SQL API features (ESQL/C)," on page 5-1. • For an Informix ESQL/C application, also follow the guidelines in Chapter 6, "IBM Informix ESQL/C features," on page 6-1.

GLS support by IBM Informix products

IBM Informix GLS supports IBM Informix products and utilities.

GLS support is provided for these IBM Informix products and utilities:

- IBM Informix database servers
- IBM Informix client applications and database server utilities
- IBM Informix GLS application programming interface

Sections that follow outline the features that GLS support provides for the first two types of IBM Informix products.

For information about how to migrate a database server whose databases contain non-English data, see the *IBM Informix Migration Guide*.

IBM Informix database servers

GLS was introduced in IBM Informix OnLine Dynamic Server.

In versions of the database server earlier than 7.0, ALS language support was provided for non-English databases with Asian (multibyte) characters and NLS language support for non-English databases with single-byte characters. GLS is a single feature that provides support for single-byte and multibyte data in non-English languages. For compatibility with earlier versions, GLS products also support all of the NLS environment variables and a subset of the ALS environment variables. For a list of these environment variables, see the *IBM Informix Migration Guide*.

If you do not install IBM Informix Client Software Development Kit or IBM Informix International Language Supplement, the following locales are installed by default:

- Chinese
  - **zh_cn**
  - **zh_hk**
  - **zh_tw**
- Eastern Europe
  - **cs_cz**
  - **pl_pl**
  - **ru_ru**
  - **sk_sk**
- Japanese
  - **ja_jp**
- Korean
  - **ko_kr**
- Thai
  - **th_th**
- Western Europe
  - **da_dk**
  - **de_at**
  - **de_ch**
  - **de_de**
  - **en_au**

- **en_gb**
- **fi_fi**
- **fr_be**
- **fr_ca**
- **fr_ch**
- **fr_fr**

The **en_us** locale is installed by the GLS-core feature.

Culture-specific features:

With the GLS feature, IBM Informix database servers provide support for culture-specific features.

The following culture-specific features are supported:

- Processing non-ASCII characters and strings

  You can use non-ASCII characters to name user-specifiable database objects, such as tables, columns, views, statements, cursors, and SPL routines, and you can use a collation order that suits local customs.

  You can also use non-ASCII characters in many other contexts. For example, you can use them to specify the WHERE and ORDER BY clauses of your SELECT statements or to sort data in NCHAR and NVARCHAR columns. You can use GLS collation features without the modification of existing code.
- Evaluation of expressions

  You can use non-ASCII characters in expression comparisons that involve any character-based data type.
- Translation of locale-specific values for dates, times, numeric data, and monetary data

  You can use end-user formats that are specific to a country or culture outside the U.S. to specify date, time, numeric, and monetary values when they are displayed in literal strings. The database server can translate these formats to the appropriate internal database format.
- Accessibility of formerly incompatible character code sets

  The client application can perform code-set conversion between convertible code sets to allow you to access and share data between databases and clients that have different code sets. For more information about code-set conversion, see "Perform code-set conversion" on page 1-29.

IBM Informix client applications and utilities

In general, a client application is a program that runs on a workstation or a PC on a network.

To the GLS feature, a *client application* can be either an IBM Informix SQL API product (such as IBM Informix ESQL/C) or an IBM Informix database server utility (such as DB-Access, **dbexport**, or **onmode**). These IBM Informix client applications support GLS:

- The DB-Access utility, which is provided with IBM Informix database servers, allows user-specifiable database objects such as tables, columns, views, statements, cursors, and SPL routines to include non-ASCII characters and to be sorted according to localized collation rules. For more information about identifiers, see "Non-ASCII characters in identifiers" on page 3-1. For general information about DB-Access, see the *IBM Informix DB-Access User's Guide*.

- The SQL APIs allow host and indicator variable names and names of user-specifiable database objects such as tables, columns, views, statements, cursors, and SPL routines to include non-ASCII characters. For more information, see Chapter 5, "General SQL API features (ESQL/C)," on page 5-1.
- Database server utilities such as **dbexport** or **onmode** allow many command-line arguments to include non-ASCII characters. For more information, see Chapter 4, "Database server features," on page 4-1.
- GLS is also a feature of IBM Informix Dynamic 4GL (Version 3.0 and higher), IBM Informix 4GL (Version 7.2 and higher), and IBM Informix SQL (Version 7.2 and higher). For details of GLS implementation, see the documentation of these IBM Informix products.

The IBM Informix GLS application programming interface

IBM Informix GLS is an application programming interface (API) that lets DataBlade module developers and Informix ESQL/C programmers develop globalized applications with a C-language interface.

The macros and functions of IBM Informix GLS provide access within an application to GLS locales, which contain culture-specific information. You can use IBM Informix GLS to write programs (or change existing programs) to handle different languages, cultural conventions, and code sets.

All IBM Informix GLS functions access the *current processing locale*, which is the locale that is currently in effect for an application. It is based on either the client locale (for Informix ESQL/C client applications and client LIBMI applications) or the server-processing locale (for DataBlade user-defined routines).

IBM Informix GLS provides macros and functions to help you perform the following globalization tasks:

- Process single-byte, multibyte, and wide characters
- Process single-byte, multibyte, and wide-character strings
- Memory management for multibyte and wide-character strings
- Convert date, time, money, and number strings to and from binary values
- Process input and output multibyte-character streams

IBM Informix client applications and database servers can access IBM Informix GLS. For applications, you link the IBM Informix GLS library to your application to perform locale-related tasks. IBM Informix database servers automatically include the IBM Informix GLS library.

Supported data types

The IBM Informix GLS feature supports SQL data types, user-defined data types, and smart large objects.

The GLS feature supports the following data types:

- SQL character data types
  - CHAR, VARCHAR, NCHAR, and NVARCHAR
  - LVARCHAR
  - DISTINCT types whose base type is one of the data types listed previously
  - TEXT and BYTE

  For information about GLS considerations for the character data types, see "Character data types" on page 3-6.
- SQL number and MONEY data types

For information about GLS considerations for number and MONEY data types, see "Numeric and monetary formats" on page 1-15.

- SQL DATE, and DATETIME data types

  For information about GLS considerations for DATE, and DATETIME data types, see "Date and time formats" on page 1-15.

- User-defined data types
  - Opaque data types
  - Complex data types
  - Distinct data types
- Smart large objects
  - BLOB
  - CLOB

  For GLS considerations regarding user-defined data types and smart large objects, see "Handle extended data types" on page 3-33.

- Informix ESQL/C character data types
  - **char**
  - **fixchar**
  - **string**
  - **varchar**
  - **lvarchar**

  For information about Informix ESQL/C data types, see the *IBM Informix ESQL/C Programmer's Manual*.

International Language Supplement

IBM Informix products include a core set of GLS locale files, including the default locale and most locales to support English, Western European, Eastern European, Asian, and African territories.

If you do not find a locale to support your language and territory, you can get additional locales in the International Language Supplement (ILS) product, which provides all available GLS locales and code-set conversion files. It also includes error messages to support several languages.

International Language Supplement lets you localize time, date, number and currency formats, character sets, and sorting orders. All of the provided locales work with IBM Informix GLS-enabled products. After following the installation instructions, set the **DBLANG** environment variable. Each user who wants to use a localized user interface file must set the environment variable **DBLANG** to point to the appropriate language msg directory.

Set **DBLANG** replacing `<codeset-hex>` with the appropriate code set that your system uses:

- C-shell: `setenv DBLANG msg/<lang>_<territory>/<codeset-hex>`
- Bourne-shell: `DBLANG=msg/<lang>_<territory>/<codeset-hex> export DBLANG`

To unset the **DBLANG** variable, enter the following command:

- C-shell: `unsetenv DBLANG`
- Bourne-shell: `unset DBLANG`

For more information about how to create customized message files, see "Locate message files" on page 1-32.

A GLS locale

In a client/server environment, both the database server and the client application must know which language the data is in to be able to process the application data correctly.

A GLS locale is a set of IBM Informix files that bring together the information about data that is specific to a given culture, language, or territory. In particular, a GLS locale can specify the following:

- The name of the code set that the application data uses
- The classification of the characters in the code set
- The collation (sorting) sequence to use for character data
- The user format for monetary, numeric, date, and time data

IBM Informix products use the following GLS files to obtain locale-related information. For more information, see Appendix A, "Manage GLS files," on page A-1.

Type of GLS file	Description
GLS locale files	Specify language, territory, writing direction, and other cultural conventions.
Code-set files	Specify how to map each logical character in a character set to a unique bit pattern.
Code-set-conversion files	Specify how to map each character in a "source" code set to corresponding characters in a "target" code set.
The `registry` file	Associates code-set names and aliases with code-set numbers that specify file names of locale files and code-set conversion files.

Each database is limited to a single locale, but different databases of the same database server can support different locales.

A single database can store character data from two or more languages that require different character sets by using the open source International Components for Unicode (ICU) implementation of the Unicode code set (**UTF-8**). This code set is available in GLS database server locales for many languages and territories. (Locales for some client-side systems also support the ICU code set UTF-8 and the ICU code sets UTF-16 and UTF-32.)

The SET COLLATION statement of Informix supports more than one localized collating order to sort NCHAR and NVARCHAR character strings.

Code sets for character data

A *character set* is one or more natural-language alphabets together with additional symbols for digits, punctuation, and diacritical marks. Each character set has at least one *code set*, which maps its characters to unique bit patterns. These bit patterns are called *code points*.

ASCII, ISO8859-1, Windows Code Page 1252, and EBCDIC are examples of code sets that support the English language.

The number of unique characters in the language determines the amount of storage that each character requires in a code set. Because a single byte can store values in the range 0 - 255, it can uniquely identify 256 characters. Most Western

languages have fewer than 256 characters and therefore have code sets made up of *single-byte characters*. When an application handles data in such code sets, it can assume that 1 byte stores 1 character.

The ASCII code set contains 128 characters. Therefore, the code point for each character requires 7 bits of a byte. These single-byte characters with code points in the range 0 - 128 are sometimes called ASCII or *7-bit characters*. The ASCII code set is a single-byte code set and is a subset of all code sets that IBM Informix products support.

If a code set contains more than 128 characters, some of its characters have code points that must set the eighth bit of the byte. These non-ASCII characters might be either of the following types of characters:

8-bit characters
The 8-bit characters are single-byte characters whose code points are 128 - 255. Examples from the ISO8859-1 code set or Windows Code Page 1252 include the non-English é, ñ, and ö characters. Only if the software is *8-bit clean* can it interpret these characters correctly. For more information, see "The GLS8BITFSYS environment variable" on page 2-7.

Multibyte characters
If a character set contains more than 256 characters, the code set must contain multibyte characters. A multibyte character might require 2 - 4 bytes of storage. Some East-Asian locales support character sets that can contain thousands of ideographic characters; GLS provides full support, for example, for the unified Chinese GB18030-2000 code set, which contains nearly 1.4 million code points. Such languages have code sets that include both single-byte and multibyte characters. These code sets are called *multibyte* code sets.

Some characters in the Japanese SJIS code set, for another example, are of 2 bytes or 3 bytes. Applications that handle data in multibyte code sets cannot assume that one character takes only 1 byte of storage.

Tip: In this publication, the term *non-ASCII characters* applies to all characters with a code point greater than 127. Non-ASCII characters include both 8-bit and multibyte characters.

IBM Informix products can support single-byte or multibyte code sets. For some examples of GLS locales that support non-ASCII characters, see "Support for non-ASCII characters" on page 1-24.

Tip: Throughout this publication, examples show how single-byte and multibyte characters are displayed. Because multibyte characters are usually ideographic (such as Japanese or Chinese characters), this publication does not use the actual multibyte characters. Instead, it uses ASCII characters to represent both single-byte and multibyte characters.

Character classes of the code set

A GLS locale groups the characters of a code set into *character classes*. Each class contains characters that have a related purpose.

GLS supports 12 classes. The contents of a character class can be language specific. For example, the *lower class* contains all alphabetic lowercase characters in a code

set. The code set of the default locale groups the letters a through z into the lower class, which also includes other lowercase characters such as á, è, î, õ, and ü.

To be globalized, your application must not assume which characters belong in a given character class. Instead, use IBM Informix GLS library functions to identify the class of a particular character.

Collation order for character data

Collation is the process of sorting character strings according to some order. The database server or the client application can perform collation.

The collating order affects the following tasks in SQL SELECT statements:

- Logical predicates in the WHERE clause

```
SELECT * FROM tab1 WHERE col1 > 'bob'
SELECT * FROM tab1 WHERE site BETWEEN 'abc' AND 'xyz'
```

- Sorted data that the ORDER BY clause creates

```
SELECT * FROM tab1 ORDER BY col1
```

- Comparisons in MATCHES and LIKE clauses

```
SELECT * FROM tab1 WHERE col1 MATCHES 'a1*'
SELECT * FROM tab1 WHERE col1 LIKE 'dog'
SELECT * FROM tab1 WHERE col1 MATCHES 'abc[a-z]'
```

For more information about how the database locale can affect the SELECT statement, see "Collation order in SELECT statements" on page 3-19.

IBM Informix database servers support two collation methods:

- Code-set order (the first-to-last order of characters in the code set)
- Localized order (if the locale defines a localized order)

Code-set order

Code-set order is the order of characters within a code set. The order of the code points in the code set determines the collating order.

For example, in the ASCII code set, A=65 and B=66. The character A always sorts before B because a code point of 65 is less than one of 66. But because a=97 and M=77, the string abc sorts after Me, which is not always the preferred result.

The database server uses code-set order to sort columns of these data types:

- CHAR
- LVARCHAR
- VARCHAR
- TEXT

All code sets that IBM Informix products support include the ASCII characters as the first 127 characters. Therefore, other characters in the code set have the code points 128 and greater. When the database server sorts values of these data types, it puts character strings that begin with ASCII characters before characters strings that begin with non-ASCII characters in the sorted results.

For an example of data sorted in code-set order, see Table 3-2 on page 3-20.

Localized order

Localized order is an order of the characters that relates to a natural language. The locale defines the order of the characters in the localized order.

For example, even though the character À might have a code point of 133, the localized order can list this character after A and before B (A=65, À=133, B=66). In this case, the string B sorts after AC but before BD.

The database server uses localized order to sort columns of these data types:

- NCHAR
- NVARCHAR

The localized order can include *equivalent characters*, those characters that the database server is to consider as equivalent when it collates them. For example, if the locale defines uppercase and lowercase versions of a character as equivalent in the localized order, then the strings Arizona, ARIZONA, and arizona are collated together, as if all three strings were the same string.

Tip: The COLLATION category of the locale file specifies the localized order, if one exists. For more information, see "The COLLATION category" on page A-3.

A localized order can also specify a collating sequence that does not match the order of code points in the character set of the locale. For example, a telephone book might require the following sort order:

```
Mabin
McDonald
MacDonald
Madden
```

A dictionary, however, might use this collating order for the same names:

```
Mabin
Madden
MacDonald
McDonald
```

If the GLS locale defines a localized order, the database server sorts data from NCHAR and NVARCHAR columns in this localized order. For an example of data sorted in a localized order, see Table 3-3 on page 3-20.

IBM Informix supports the SET COLLATION statement, which can specify a localized collation different from the **DB_LOCALE** setting. The scope of the non-default collating order is the current session, but database objects that perform collation, such as indexes or triggers, use the collating order from the time of their creation when they sort NCHAR or NVARCHAR values.

After the SET COLLATION statement has specified a localized collation order, and you have completed all of the sorting tasks that require that localized order, you can restore the collation that the **DB_LOCALE** setting implies by issuing the SET NO COLLATION statement of SQL.

The SET COLLATION statement only affects localized collation operations that the database server performs. Sorting of NCHAR or NVARCHAR data values by the client always follows the collation order of the **CLIENT_LOCALE** setting, and ignores any SET COLLATION specifications. For more information about the environment variables that can define the client locale or the server locale, see "Locales in the client/server environment" on page 1-16.

Unicode collation

The open source International Components for Unicode (ICU) implementation of the Unicode code set (UTF-8) is available in GLS locales for many languages and territories.

For example, the **en_us.utf8** locale supports the Unicode code set. The GLS 6.00 library incorporates the International Components for Unicode (ICU) 4.8.1 library. For more information about ICU, see the ICU website at http://www.ibm.com/software/globalization/icu/index.jsp.

GLS locales that use the Unicode code set (UTF-8) support Unicode collation of NCHAR and NVARCHAR data by the ICU Unicode Collation Algorithm. For more information about this algorithm, see the Unicode website at http://www.unicode.org/unicode/reports/tr10.

Important: If Unicode produces index keys that are too long to fit in the default dbspace page size, use a larger, nondefault page size.

Collation support

Collation by IBM Informix database servers depends on the data type of the database column.

The following table summarizes the collation rules.

Column data types	Collating order
CHAR, VARCHAR, TEXT	Code-set order
LVARCHAR	Code-set order
NCHAR, NVARCHAR	Localized order

The difference in collation is the only distinction between the CHAR and NCHAR data types and between the VARCHAR and NVARCHAR data types. For more information about collation, see "Character data types" on page 3-6. If a locale does not define a localized order, the database server collates NCHAR and NVARCHAR data values in code-set order.

Important: There is an exception to the general rule that CHAR, LVARCHAR, and VCHAR values are always sorted in the code-set order. The MATCHES operator always uses the localized order, if one is defined, to evaluate range expressions for character values, regardless of the data type. See "MATCHES condition" on page 3-24.

End-user formats

The *end-user format* is the format in which a data value is displayed or entered in a client application as a literal string or a character variable.

An end-user format is useful for a data type whose format in the database is different from the format to which users are accustomed. The database server stores data for DATE, DATETIME, MONEY, and numeric data types in compact internal formats within the database.

For example, the database server stores a DATE value as an integer number of days since December 31, 1899, so the date `03/19/96` is `35142`. This internal format is not intuitive.

IBM Informix products support end-user formats so that a client application can use this more intuitive form instead of the internal format. Literal strings or character variables can show in SQL statements as column values or as arguments of SQL API library functions.

An IBM Informix product uses an end-user format when it encounters a string (a literal string or the value in a character variable) in these contexts:

- When an IBM Informix product reads a string, it uses an end-user format to determine how to interpret the string so that it can convert it to a numeric value.

  For example, suppose that DB-Access has the default (U.S. English) as its client locale. The literal date in the following INSERT statement uses the end-user format for dates that the default locale defines:

  ```
  INSERT INTO mytab ( date1 ) VALUES ( '03/19/96' )
  ```

  When it receives the data from the client application, the database server uses the end-user format to interpret this literal date so that it can convert it to the corresponding internal format (35142).
- When an IBM Informix product prints a string, it uses an end-user format to determine how to format the numeric value as a string.

  For example, suppose that an Informix ESQL/C client application has a French locale as its client locale, and this locale defines a date end-user format that formats dates as dd/mm/yy. The following **rdatestr()** function uses the end-user format for dates to obtain the value in the **datestr** character variable:

  ```
  err = rdatestr(jdate, datestr);
  ```

  The **rdatestr()** function uses the end-user format to determine how to format the internal format (35142) as a date string before it puts the value in the **datestr** variable. For more information about the effect of the GLS feature on SQL API library functions, see "Enhanced ESQL/C library functions" on page 6-7.

A GLS locale defines end-user formats for the following types of data:

- Representation of currency notation and numeric format
- Representation of dates and of time-of-day values

You can specify number, currency, date, and time values in an end-user format that is specific to a given country or culture.

Important: End-user formats of date, time, number, and monetary values do not affect the internal format of the corresponding data types in the database. They affect only how the client application displays the data and interprets data entry.

The following table lists the values that define the end-user format for each data type that uses end-user formats. For information about the environment variables, see Chapter 2, "GLS environment variables," on page 2-1. For information about the locale categories, see Appendix A, "Manage GLS files," on page A-1.

Data types	Environment variables	Locale category
DATE	**GL_DATE**	TIME
DATETIME	**GL_DATE**	TIME
INTERVAL	**GL_DATETIME**	
MONEY	**DBMONEY**	MONETARY

Data types	Environment variables	Locale category
Number (DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INT8, INTEGER, NUMERIC, REAL, SMALLFLOAT, SMALLINT)	None	NUMERIC

Numeric and monetary formats

When an IBM Informix product reads a string that contains numeric or monetary data, it uses the end-user format to determine how to convert this string to the internal value for the database column.

When an IBM Informix product prints a string that contains numeric or monetary data, it uses the end-user format to determine how to format the internal value for the database column as a string.

End-user formats for numbers and currency specify these elements:

- The *decimal-separator symbol* separates the part of the numeric value from the fractional part. In the default locale, the period is the decimal separator (3.01). In a locale such as French, the comma is the decimal separator (3,01).
- The *thousands-separator symbol* can show between groups of digits in the part of the numeric value. In the default locale, the comma is the thousands separator (3,255); in a French locale, the space is the thousands separator (3 255).
- The characters that indicate positive and negative numbers.
- The number of digits to group between each appearance of a non-monetary thousands separator.

For example, this might specify that numbers always omit the separator after the millions position, which produces the following output: 1234,345.

In addition to this notation, monetary data also uses a currency symbol to identify the currency unit. This can show at the front ($100) or back (100FF) of the monetary value. In this publication, the combination of currency symbol, decimal separator, and thousands separator is called *currency notation*.

Date and time formats

When an IBM Informix product reads a string that contains time data, it uses the end-user format to determine how to convert this string to the internal integer value for a DATETIME column.

When an IBM Informix product prints a string that contains time data, it uses the time end-user format to determine how to format the internal integer value for a DATETIME column as a string. In the same way, IBM Informix products use the date end-user format to read and print strings for the internal values of the date data types.

Important: End-user formats specify how client applications view data, but do not affect the internal format of DATETIME or DATE values stored in the database.

The end-user formats for date and time can include the names and abbreviations for days of the week and months of the year, and the commonly used representations for dates, time (12-hour and 24-hour), and DATETIME values.

End-user formats can include names of eras (as in the Japanese Imperial date system) and non-Gregorian calendars (such as the Arabic lunar calendar).

For example, the Taiwan culture uses the Ming Guo year format in addition to the Gregorian calendar year. For dates before 1912, Ming Guo years are negative. The Ming Guo year 0000 is undefined; any attempt to use it generates an error. The following table shows some era-based dates.

Gregorian year	Ming Guo year	Remarks
1993	82	1993 – 1911 = 82
1912	01	1912 – 1911 = 01
1911	–01	1911 – 1912 = –01
1910	–02	1910 – 1912 = –02
1900	–12	1900 – 1912 = –12

Japanese Imperial-era dates are tied to the reign of the Japanese emperors. The following table shows Julian and Japanese era dates. It shows the Japanese era format in full, with abstract multibyte characters for the Japanese characters, and in an abbreviated form that uses romanized characters (gengo). The abbreviated form of the era uses the first letter of the English name for the Japanese era. For example, `H` represents the Heisei era.

Gregorian date	Abstract Japanese era (in full)	Japanese era (gengo)
1868/09/08	$A^1A^2B^1B^2$01/09/08	M01/09/08
1912/07/30	$A^1A^2B^1B^2$45/07/30	M45/07/30
1912/07/31	$A^1A^2B^1B^2$01/07/31	T01/07/31
1926/12/25	$A^1A^2B^1B^2$15/12/25	T15/12/25
1926/12/26	$A^1A^2B^1B^2$01/12/26	S01/12/26
1989/01/07	$A^1A^2B^1B^2$64/01/07	S64/01/07
1989/01/08	$A^1A^2B^1B^2$01/01/08	H01/01/08
1995/01/01	$A^1A^2B^1B^2$07/01/01	H07/01/01

Here A^1A^2 and B^1B^2 represent multibyte Japanese characters. For more information, see "Customize date and time end-user formats" on page 1-32.

Set a GLS locale

For the database server and the client application to communicate successfully, you must establish the appropriate GLS locales for your environment.

A GLS locale name identifies the language, territory, and code set that you want your IBM Informix product to use. For the syntax of the components of locale names, see "The CLIENT_LOCALE environment variable" on page 2-2.

IBM Informix products use the locale name to find the corresponding *locale files*. A locale file is a runtime version of the locale information. The locale name must correspond to a GLS locale file in a subdirectory of the IBM Informix installation directory (which **INFORMIXDIR** specifies) called `gls`. For more information about GLS locale files, see Appendix A, "Manage GLS files," on page A-1.

Locales in the client/server environment

In a client/server environment, the client application, database server, and one or more databases might be on different computers.

The following figure shows an example of database server connections between an IBM Informix ESQL/C client application and the **acctng** database through an IBM Informix database server.

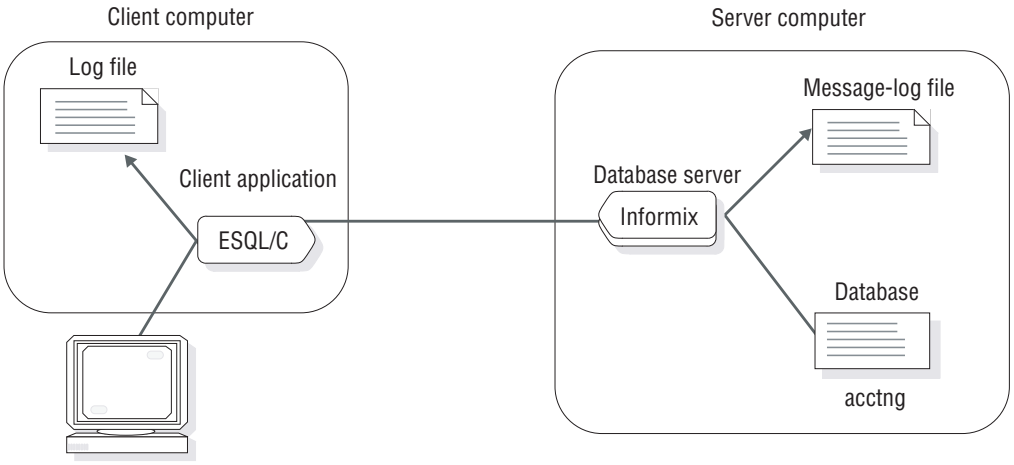


Figure 1-1. Example of a client/server environment

These computers might have different operating systems or different language support. To ensure that these three parts of the database application communicate locale information successfully, IBM Informix products support the following locales:

- The *client locale* identifies the locale that the client application uses.
- The *database locale* identifies the locale of the data in a database.
- The *server locale* identifies the locale that the database server uses for its server-specific files.

The following figure shows the client locale, database locale, and server locale that the example Informix ESQL/C application (from the previous figure) establishes.

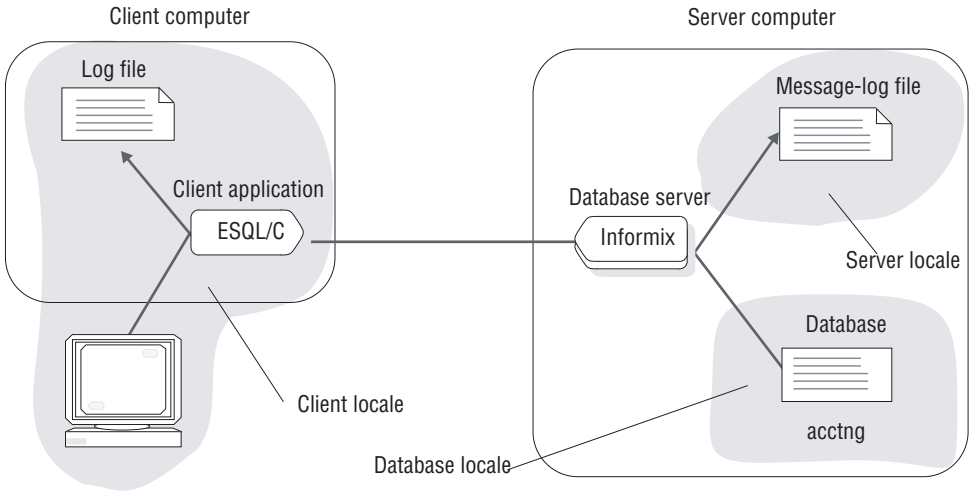


Figure 1-2. The client locale, database locale, and server locale

When you set the same or compatible GLS locales for each of these locales, your client application is not dependent on how the operating system of each computer implements language-specific features.

Sections that follow describe each of these locales in more detail.

The client locale

The *client locale* specifies the language, territory, and code set that the client application uses to perform read and write (I/O) operations.

In a client application, I/O operations include reading a keyboard entry or a file for data to be sent to the database and writing data that the database server retrieves from the database to the screen, a file, or a printer. In addition, an SQL API client uses the client locale for literal strings (end-user formats), embedded SQL (ESQL) statements, and host variables.

IBM Informix products use the **CLIENT_LOCALE** environment variable for the following purposes:

- When the preprocessor for Informix ESQL/C processes a source file, it accepts C source code that is written in the code set of the **CLIENT_LOCALE**.

  The C compiler and the operating system that you use might impose limitations on the Informix ESQL/C program. For more information, see "Generate non-ASCII file names" on page 6-3.
- When an Informix ESQL/C client application executes, it checks **CLIENT_LOCALE** for the name of the client locale, which affects operating-system file names, contents of text files, and formats of date, time, and numeric data.

  For more information, see "Handle non-ASCII characters" on page 6-1.
- When a client application and a database server exchange character data, the client application performs code-set conversion when the code set of the **CLIENT_LOCALE** environment variable is different from the code set of **DB_LOCALE** (on the client computer).

  Code-set conversion prevents data corruption when these two code sets are different. For more information, see "Perform code-set conversion" on page 1-29.
- When the client application requests a connection, it sends information, including the **CLIENT_LOCALE**, to the database server.

  The database server uses **CLIENT_LOCALE** when it determines how to set the client-application information of the server-processing locale. For more information, see "Establish a database connection" on page 1-24.
- When database utilities create files, the file names and file contents are in the code set that **CLIENT_LOCALE** specifies.
- When a client application looks for product-specific message files, it checks the message directory associated with the client locale.

  For more information, see "Locate message files" on page 1-32.

In the example connection that Figure 1-2 on page 1-17 shows, if the client locale is German with the Windows Code Page 1252 (**de_de.1252@euro**), the German locale-specific information that the Informix ESQL/C client application uses includes the following:

- Valid date end-user formats support the following format for the U.S. English date of Tuesday, 02/11/1997:

  ```
  Di., 11. Feb 1997
  ```
- Valid monetary end-user formats support the following format for the U.S. English amount of $354,446.02:

  ```
  EUR354.446,02
  ```

Tip: To provide this information for the client locale, the locale file contains the following locale categories: COLLATION, CTYPE, TIME, MONETARY, and NUMERIC. For more information, see "Locale categories" on page A-2.

To determine the client locale, client applications use environment variables set on the client computer. To obtain the localized order and end-user formats of the client locale, a client application uses the following precedence:

1. **DBDATE** and **DBTIME** environment variables for the end-user formats of date and time data and **DBMONEY** for the end-user format of monetary data (if one of environment variables is set)
2. **GL_DATE** and **GL_DATETIME** environment variables for the end-user formats of date and time data (if one of environment variables is set)
3. The information that the client locale defines (**CLIENT_LOCALE**, if it is set)
4. The default locale (U.S. English)

Client applications that are based on IBM Informix use the precedence of steps 2, 3, and 4 in the preceding list. You do not need to set the other environment variables for Informix client applications.

Support for **DBDATE** and **DBTIME** provides compatibility with earlier versions for client applications based on earlier versions of Informix products. It is recommended that you use **GL_DATE** and **GL_DATETIME** for new applications.

The database locale

The *database locale*, which is set with the **DB_LOCALE** environment variable, specifies the language, territory, and code set that the database server needs to correctly interpret locale-sensitive data types (NCHAR and NVARCHAR) in a particular database.

The code set specified in **DB_LOCALE** determines which characters are valid in any character column and the names of database objects such as databases, tables, columns, and views. For more information, see "Name database objects" on page 3-1.

The database locale also specifies the writing direction. Most languages are written left-to-right, but some are written right-to-left or top-to-bottom.

IBM Informix products use the **DB_LOCALE** environment variable for the following purposes:

- When a client application and a database server exchange character data, the client application performs code-set conversion when the value of the **DB_LOCALE** environment variable (on the client computer) is different from the value of **CLIENT_LOCALE**.

  Code-set conversion prevents data corruption when these two code sets are different. For more information, see "Perform code-set conversion" on page 1-29.
- When the client application requests a connection, it sends information, including the **DB_LOCALE** (if it is set), to the database server.

  The database server uses **DB_LOCALE** when it determines how to set the database information of the server-processing locale. For more information, see "Establish a database connection" on page 1-24.
- When a client application tries to open a database, the database server compares the value of the **DB_LOCALE** environment variable that the client application passes with the database locale that is stored in the database.

When the database server accesses columns of locale-sensitive data types, it uses the locale that **DB_LOCALE** specifies. For more information, see "Verify the database locale" on page 1-25.

- When the database server creates a database, it examines the database locale (**DB_LOCALE**) to determine how to store character information in the system catalog of the database. This information includes operations such as how to handle regular expressions, compare character strings, and ensure correct use of code sets.

The database server stores a condensed version of the database locale in the **systables** system catalog table.

When the database server stores the database locale information directly in the system catalog, it permanently attaches the locale to the database. This information is used throughout the lifetime of the database. In this way, the database server can always determine the locale that it needs to interpret the locale-sensitive data correctly.

The SET COLLATION statement can specify the localized collation of a different locale to sort NCHAR and NVARCHAR data in the current session.

The condensed version of the database locale is stored in the following two rows of **systables**, which store the condensed locale name in the **site** column:

- The row with **tabid** `90` stores the COLLATION category of the database locale. The collation order determines the order in which the characters of the code set collate. If the database locale defines only a code-set order for collation (as does the default locale, U.S. English), the database server creates CHAR and VARCHAR columns to store the character information. If the database locale defines a localized order for collation, however, the database server creates NCHAR and NVARCHAR columns to store this character information. The **tabname** value for this row is GLS_COLLATE.
- The row with **tabid** `91` stores the CTYPE category of the database locale. The CTYPE category of a locale determines how characters of the code set are classified. The database server uses character classification for case conversion and some regular-expression evaluation. The **tabname** value for this row is GLS_CTYPE.

The database server uses the value of the **DB_LOCALE** environment variable that the client application sends. If you do not set **DB_LOCALE** on the client computer, however, the database server uses the value of **DB_LOCALE** on the server computer as the database locale.

In the connection shown in Figure 1-2 on page 1-17, the database server references the database locale when the client application requests sorted information for an NCHAR column in the **acctng** database. If this locale is German with the Windows Code Page 1252 (**de_de.1252**), the database server uses a localized order that sorts accented characters, such as `ö`, after their unaccented counterparts. Thus, the string `öff` sorts after `ord` but before `pre`. For the syntax to set the database locale, see "The DB_LOCALE environment variable" on page 2-4.

The server locale

The *server locale*, which is set with the **SERVER_LOCALE** environment variable, specifies the language, territory, and code set that the database server uses to perform read and write (I/O) operations on the server computer (the computer on which the database server runs).

These I/O operations include reading or writing the following files:

- Diagnostic files that the database server generates to provide additional diagnostic information
- Log files that the database server generates to record events
- The explain output file that the SQL statement SET EXPLAIN generates

The database server does not use the server locale, however, to write files that are in an IBM Informix proprietary format (database and table files). For a more detailed description of the files that the database server writes by using the server locale, see Chapter 4, "Database server features," on page 4-1.

The database server looks for product-specific message files in the message directory that is associated with the locale specified in **SERVER_LOCALE**. For more information, see "Locate message files" on page 1-32.

In the example connection that Figure 1-2 on page 1-17 shows, the IBM Informix database server uses the locale specified in **SERVER_LOCALE** to determine the code set to use when it writes a message-log file. For the syntax to set the server locale, see "The SERVER_LOCALE environment variable" on page 2-21.

Tip: The database server is the only IBM Informix product that needs to know the server locale. Any database server utilities that you run on the server computer use the client locale to read from and write to files and the database locale (on the server computer) to access databases that are set on the server computer.

The server locale and the server-processing locale are two different locales. For more information about the server-processing locale, see "Determine the server-processing locale" on page 1-26.

Related concepts:

Using the FILE TO option (SQL Syntax)

Default name and location of the explain output file on UNIX (SQL Syntax)

Default name and location of the output file on Windows (SQL Syntax)

Related reference:

onmode -Y: Dynamically change SET EXPLAIN (Administrator's Reference)

onmode and Y arguments: Change query plan measurements for a session (SQL administration API) (Administrator's Reference)

SET EXPLAIN statement (SQL Syntax)

The default locale

IBM Informix products use U.S. English as the *default locale* if you do not set the environment variables that can specify a locale.

The default locale specifies the following information:

- The U.S. English language and an English-language code set
- Standard U.S. formats for monetary, numeric, date, and time data

To use the default locale for database applications requires no special steps. To use a customized version of U.S. English, British English, or another language, however, your environment must identify the appropriate locale.

For information about how to specify a GLS locale, see "Set a nondefault locale" on page 1-23.

The default code set

The *default code set* is the code set that the default locale supports. When you use the default locale, the default code set supports both the ASCII code set and some set of 8-bit characters.

The default locale, U.S. English, has the following locale name, where `en` indicates the English language, `us` indicates the United States territory, and the numbers indicate the platform-specific name of the default code set.

For a chart of ASCII values, see the Relational Operator segment in the *IBM Informix Guide to SQL: Syntax*. The following table describes the default code set for UNIX and for Windows platforms.

Platform	Default code set
UNIX	ISO8859-1
Windows	Microsoft 1252

In a locale name, you can specify the code set as either the code-set name or the condensed form of the code-set name. For example, the following locale names both identify the U.S. English locale with the ISO8859-1 code set:

- UNIX

  The locale name **en_us.8859-1** uses the code-set name to identify the ISO8859-1 code set.
- Windows

  The locale name **en_us.0333** uses the condensed form of the code-set name to identify the ISO8859-1 code set.

For more information about the condensed form of a code-set name, see "Code-set-conversion file names" on page A-9.

Default end-user formats for date and time

In the default locale, IBM Informix products use end-user formats for date and time values.

IBM Informix products use the following user formats:

- For DATE values: `%m/%d/%iy`
- For DATETIME values: `%iY-%m-%d %H:%M:%S`

For information about these formatting directives, see "The GL_DATE environment variable" on page 2-10 and "The GL_DATETIME environment variable" on page 2-15. For an introduction to date and time end-user formats, see "Date and time formats" on page 1-15. For information about how to customize these end-user formats, see "Customize date and time end-user formats" on page 1-32.

Default end-user formats for numeric and monetary values

When you use the default locale, IBM Informix products use end-user formats for numeric and monetary values.

IBM Informix products use the following end-user formats:

- The thousands separator is the comma (,).

- The decimal separator is the period (.).
- Three digits show between each thousands separator.
- The positive sign is plus (+) and the negative sign is minus (-).

For monetary values, IBM Informix products also use a currency symbol, the dollar ($) sign, in front of a monetary value. For an introduction to numeric and monetary end-user formats, see "Numeric and monetary formats" on page 1-15. For information about how to customize these end-user formats, see "Customize monetary values" on page 1-34.

Set a nondefault locale

By default, IBM Informix products use the U.S. English locale, but IBM Informix products support many other locales.

To use a nondefault locale, you must set the following environment variables:

- Set the **CLIENT_LOCALE** environment variable to specify the appropriate client locale.

  If you do not set **CLIENT_LOCALE**, the client locale is the default locale, U.S. English.
- Set **DB_LOCALE** on each client computer to specify the database locale for a client application to use when it connects to a database.

  If you do not set **DB_LOCALE** on the client system, the client application sets **DB_LOCALE** to the client locale. This default value avoids the need for the client application to perform code-set conversion.

  You might also want to set **DB_LOCALE** on the server computer so that the database server can perform operations such as the creation of databases (when the client does not specify its own **DB_LOCALE**).
- Set the **SERVER_LOCALE** environment variable to specify the appropriate server locale.

  If you do not set **SERVER_LOCALE**, the server locale is the default locale, U.S. English.

To access a database that has a nondefault locale, the **CLIENT_LOCALE** and **DB_LOCALE** settings on the client system must support this nondefault locale. Both locales must be the same, or their code sets must be convertible, as described in "Perform code-set conversion" on page 1-29.

For example, to access a database with a Japanese SJIS locale, set both **DB_LOCALE** and **CLIENT_LOCALE** to **ja_jp.sjis** on the client system. (If you set **DB_LOCALE** but not **CLIENT_LOCALE**, the client application returns an error, because it cannot set up code-set conversion between the SJIS database code set and the code set of the default locale on the client system.)

When a client application requests a connection, the database server uses information in the client, database, and server locales to create the server-processing locale. For more information, see "Establish a database connection" on page 1-24.

GLS locales with IBM Informix products

IBM Informix products use GLS locales for several tasks.

These tasks include:

- When a client application requests a connection, the database server uses the client and database locales to determine if these locales are compatible.
- When a client application first begins execution, it compares the client and database locales to determine if it must perform code-set conversion.
- All IBM Informix products that show product-specific messages look in a directory specific to the client locale to find these messages.

Support for non-ASCII characters

An IBM Informix product obtains its code set from its GLS locale. Locales are available for both single-byte and multibyte code sets.

All supported code sets define the ASCII characters. Most also support additional non-ASCII characters (8-bit or multibyte characters). For more information about code sets and non-ASCII characters, see "Code sets for character data" on page 1-9.

The following types of GLS locales are examples of locales that contain non-ASCII characters in their code sets:

- The default locale supports the default code set, which contains 8-bit characters for non-English characters such as é, ñ, and ö.

  The name of the default code set depends on the platform on which your IBM Informix product is installed. For more information about the default code set, "The default code set" on page 1-22.
- Many nondefault locales support the default code set.

  Nondefault locales that support the UNIX default code set, ISO8859-1, include British English (**en_gb.8859-1**), French (**fr_fr.8859-1**), Spanish (**es_es.8859-1**), and German (**de_de.8859-1**).
- Other nondefault locales, such as Japanese SJIS (**ja_jp.sjis**), Korean (**ko_kr.ksc**), and Chinese (**zh_cn.gb**), contain multibyte code sets. (The unified Chinese code set is GB18030-2000.)

For the contexts in which you can use non-ASCII characters, including multibyte characters, see Chapter 3, "SQL features," on page 3-1, Chapter 4, "Database server features," on page 4-1, and Chapter 5, "General SQL API features (ESQL/C)," on page 5-1.

For an IBM Informix product to support non-ASCII characters, however, it must use a locale that supports a code set with the same non-ASCII characters.

Establish a database connection

To establish a database connection, the GLS locales performs a series of steps.

When a client application requests a connection to a database, the database server uses GLS locales to perform the following steps:

1. Examine the client locale information that the client passes.
2. Verify that it can establish a connection between the client application and the database that it requested.
3. Determine the server-processing locale, which the database server uses to handle locale-specific information for the connection.

Send the client locale

When the client application requests a connection, it sends the environment variables from the client locale to the database server.

The following environment variables are sent from the client locale to the database server:

- Locale information
  - **CLIENT_LOCALE**

    If **CLIENT_LOCALE** is not set, the client sets it to the default locale.
  - **DB_LOCALE**

    If **DB_LOCALE** is not set, the client does not send a **DB_LOCALE** value to the database server.
- User-customized end-user formats
  - Date and time end-user formats: **GL_DATE** and **GL_DATETIME**
  - Monetary end-user formats: **DBMONEY**

  If you do not set any of these environment variables, the client application does not send them to the database server, and the database server uses the end-user formats that the **CLIENT_LOCALE** defines.

The database server uses these settings to extract the following information:

- How are numeric and monetary values formatted?
- How are dates and times formatted?
- What database locale does the client expect?

The database server uses this information to verify the database locale and to establish the server-processing locale.

Verify the database locale

To open an existing database, the client application must correctly identify the database locale for that database.

To verify the database locale, the database server compares the following two locales:

- The locale specified by **DB_LOCALE** that the client application sends
- The database locale that is stored in the system catalog of the database that the client application requests.

  For more information, see "The database locale" on page 1-19.

Two database locales match if their language, territory, code set, and any locale modifiers are the same. If these database locales do not match, the database server performs the following actions:

- It sets the eighth character field of the SQLWARN array in the SQL Communications Area (SQLCA structure) to `W` as a warning flag. Values for `W` are ASCII 32 (blank) and ASCII 87 (W).
- It uses the database locale that is stored in the system catalog of the requested database as the database locale.

Important: Check for the SQLWARN warning flag after your client application requests a connection. If the two database locales do not match, the client application might incorrectly interpret data that it retrieves from the database server, or the database server might incorrectly interpret data that it receives from the client. If you proceed with such a connection, it is your responsibility to understand the format of the data that is being exchanged.

Check for connection warnings

To check for the eighth character field of the SQLWARN array, an IBM Informix ESQL/C client application can check the **sqlca.sqlwarn.sqlwarn7** field.

If the **sqlwarn7** field has a value of `W`, the database server has ignored the database locale that the client specified and has instead used the locale in the database as the database locale.

For more information about how to handle exceptions within an ESQL program, see the *IBM Informix ESQL/C Programmer's Manual*.

Important: Array elements in SQLWARN arrays are numbered starting with zero in IBM Informix ESQL/C, but starting with one in other languages. For IBM Informix GLS tools that use one-based counts on arrays, such as IBM Informix 4GL and IBM Informix Dynamic 4GL, the warning character that IBM Informix ESQL/C calls **sqlca.sqlwarn.sqlwarn7** is called **SQLCA.SQLAWARN[8]**.

Determine the server-processing locale

The database server uses the *server-processing locale* to obtain locale information for its own internal sessions and for any connections.

When the database server begins execution, it initializes the server-processing locale to the default locale. When a client application requests a connection, the database server must redetermine the server-processing locale to include the client and database locales. The database server uses the server-processing locale to obtain locale information that it needs when it transfers data between the client system and the database.

After the IBM Informix database server verifies the database locale, it uses a precedence of environment variables from the client and database locales to set the server-processing locale.

The database server obtains the following information from the server-processing locale:

- Locale information for the database

  This database information includes the localized order and code set for data in columns with the locale-sensitive data types (NCHAR and NVARCHAR). The database server obtains this information from the name of the database locale that it has verified.
- Locale information for client-application data

  This client-application information provides the end-user formats for date, time, numeric, and monetary data. The database server obtains this information from the client application when the client requests a connection.

The following figure shows the relationship between the client locale, database locale, server locale, and server-processing locale.

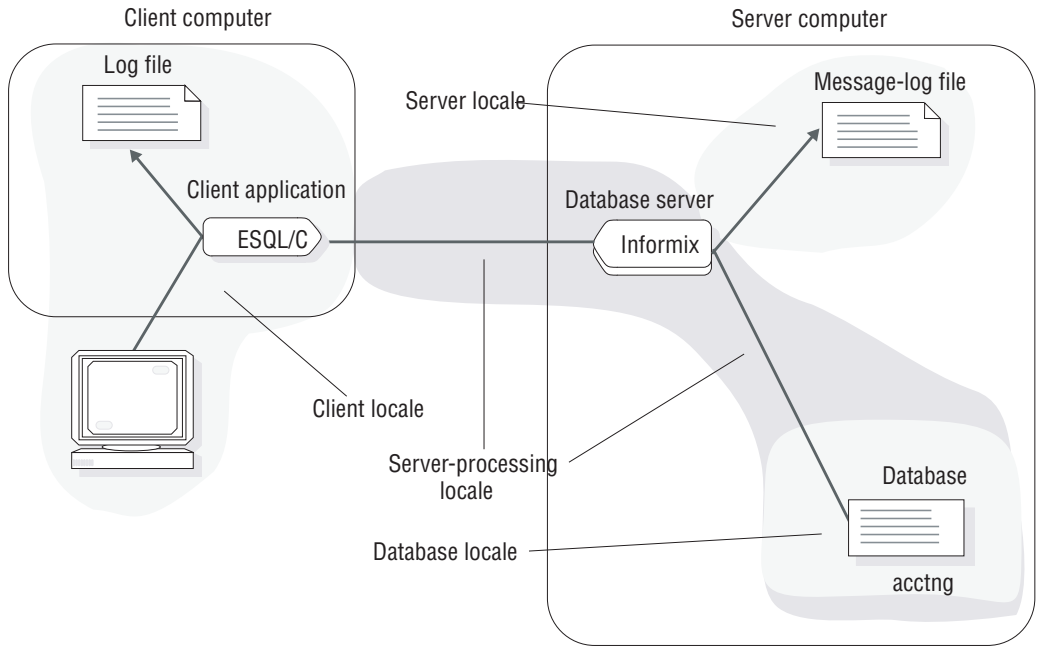


Figure 1-3. The server-processing locale

Tip: The database server uses the server locale, as specified by the **SERVER_LOCALE** environment variable, for read and write operations on its own operating-system files. For information about operating-system files, see "GLS support by IBM Informix database servers" on page 4-1.

Locale information for the database:

The database server must know how to interpret the data in any columns with the locale-specific data types, NCHAR and NVARCHAR.

To handle this locale-specific data correctly, the database server must know the localized order for the collation of the data and the code set of the data. In addition, the database server uses the code set of the database locale as the code set of the server-processing locale.

The database server might have to perform code-set conversion between the code sets of the server-processing locale and the server locale. For more information, see "Perform code-set conversion" on page 1-29.

The database server uses the following precedence to determine this database information:

1. The locale that the database server uses to determine the database information for the server-processing locale depends on the state of the database to which the client application requests a connection, as follows:
   a. For a connection to an existing database, the database server uses the database information from the database locale that it obtains when it verifies the database locale. If the client application does not send **DB_LOCALE**, the database server uses the **DB_LOCALE** that is set on the server computer.
   b. For a new database, the database server uses the **DB_LOCALE**, which the client application has sent.

2. The locale that the **DB_LOCALE** environment variable on the server computer indicates
3. The default locale (U.S. English)

IBM Informix uses the precedence of steps 1 on page 1-27, 2, and 3 in the preceding list to obtain the database information for the server-processing locale. You are not required to set the other environment variables.

Tip: The precedence rules apply to how the database server determines both the COLLATION category and the CTYPE category of the server-processing locale. For more information about these locale categories, see "Locale categories" on page A-2.

For more information about how the database server obtains these environment variables, see "Send the client locale" on page 1-24.

If the client application makes another request to open a database, the database server must reestablish the database information for the server-processing locale, as follows:

1. Reverify the database locale by comparing the database locale in the database to be opened with the value of the **DB_LOCALE** environment variable from the client application.
2. Reestablish the server-processing locale with the newly verified database locale (from the preceding step).

For example, suppose that your client application has **DB_LOCALE** set to **en_us.8859-1** (U.S. English with the ISO8859-1 code set). The client application then opens a database with the U.S. English locale (**en_us.8859-1**), and the database server establishes a server-processing locale with **en_us.8859-1** as the locale that defines the database information.

If the client application now closes the U.S. English database and opens another database, one with the French locale (**fr_fr.8859-1**), the database server must reestablish the server-processing locale. The database server sets the eighth character field of the SQLWARN array to `W` indicate that the two locales are different. The client application, however, might choose to use this connection because both these locales support the ISO8859-1 code set. If the client application opens a database with the Japanese SJIS locale (**ja_jp.sjis**) instead of one with a French locale, your client application would probably not continue with this connection because the locales are too different.

Locale information for the client application:

The database server must know how to interpret the end-user formats when they show in monetary, date, or time data that the client application sends. It must also convert data from the database to any appropriate end-user format before it sends this data to the client application.

For more information about end-user formats, see "End-user formats" on page 1-13.

The database server uses the following precedence to determine this client-application information:

1. **DBDATE** and **DBTIME** environment variables for the date and time end-user formats and **DBMONEY** for the monetary end-user formats (if one of environment variables is set on the client)

   Support for **DBDATE** and **DBTIME** provides compatibility with earlier versions for client applications that are based on earlier versions of IBM Informix products. It is recommended that you use **GL_DATE** and **GL_DATETIME** for new applications.
2. **GL_DATE** and **GL_DATETIME** environment variables (if one of environment variables is set on the client) for the date and time end-user formats
3. The locale that the **CLIENT_LOCALE** environment variable from the client application indicates

Tip: The precedence rules apply to how the database server determines the NUMERIC, MONETARY, TIME, and MESSAGES categories of the server-processing locale. For more information about these locale categories, see "Locale categories" on page A-2.

The client application passes the **DBDATE**, **DBMONEY**, **DBTIME**, **GL_DATE**, and **GL_DATETIME** environment variables (if they are set) to the database server. It also passes the **CLIENT_LOCALE** and **DB_LOCALE** environment variables. For more information, see "Send the client locale" on page 1-24.

Perform code-set conversion

In a client/server environment, character data might need to be converted from one code set to another if the client or server computer uses different code sets to represent the same characters. The conversion of character data from one code set (the source code set) to another (the target code set) is called *code-set conversion*.

Without code-set conversion, one computer cannot correctly process or show character data that originates on the other (when the two computers use different code sets).

IBM Informix products use GLS locales to perform code-set conversion. Both an IBM Informix client application and a database server might perform code-set conversion. For details, see "Database server code-set conversion" on page 4-2 and "Client application code-set conversion" on page 5-1.

You specify a code set as part of the GLS locale. At runtime, IBM Informix products adhere to the following rules to determine which code sets to use:

- The client application uses the *client code set*, which the **CLIENT_LOCALE** environment variable specifies, to write all files on the client computer and to interact with all client I/O devices.
- The database server uses the *database code set*, which the **DB_LOCALE** environment variable specifies, to transfer data to and from the database.
- The database server uses the *server code set*, which the **SERVER_LOCALE** environment variable specifies, to write files (such as debug and warning files).

Code-set conversion does not provide either of the following capabilities:

- Code-set conversion is not a semantic translation.

  It does not convert between words in different languages. For example, it does not convert from the English word `yes` to the French word `oui`. It only ensures that each character retains its meaning when it is processed or written, regardless of how it is encoded.

- Code-set conversion does not create a character in the target code set if it exists only in the source code set.

  For example, if the character â is passed to a target computer whose code set does not contain that character, the target computer cannot process or print the character exactly.

For each character in the source code set, a corresponding character in the target code set should exist. However, if the source code set contains characters that are not in the target code set, the conversion must then define how to map these mismatched characters to the target code set. (Absence of a mapping between a character in the source and target code sets is often called a *lossy error*.) If all characters in the source code set exist in the target code set, mismatch handling does not apply.

A code-set conversion uses one of the following four methods to handle mismatched characters:

Round-trip conversion
: This method maps each mismatched character to a unique character in the target code set so that the return mapping maps the original character back to itself. This method guarantees that a two-way conversion results in no loss of information; however, data that is converted just one way might prevent correct processing or printing on the target computer.

Substitution conversion
: This method maps all mismatched characters to one character in the target code set that highlights mismatched characters. This method guarantees that a one-way conversion clearly shows the mismatched characters; however, a two-way conversion results in loss of information if mismatched characters are present.

Graphical-replacement conversion
: This method maps each mismatched character to a character in the target code set that looks like the source character.

  This method includes the mapping of one-character ligatures to their two-character equivalents and vice versa, to make printing of mismatched data more accurate on the target computer, but it most likely confuses the processing of this data on the target computer.

A hybrid of two or three of the preceding conversion methods

Tip: Each code-set-conversion source file (`.cv`) indicates how the associated conversion handles mismatched characters. For information about code-set-conversion files, see Appendix A, "Manage GLS files," on page A-1.

When code-set conversion is performed

An application must use code-set conversion only if the two code sets (client and server-processing locale, or server-processing locale and server) are different.

The following situations are possible causes of code sets that differ:

- Different operating systems might encode the same characters in different ways.

  For example, the code for the character â (a-circumflex) in Windows Code Page 1252 is hexadecimal 0xE2. In IBM Coded Character Set Identifier (CCSID) 437 (a common IBM UNIX code set), the code is hexadecimal 0x83. If the code for â on the client is sent unchanged to the IBM UNIX computer, it prints as the Greek character g (gamma). This action occurs because the code for g is hexadecimal 0xE2 on the IBM UNIX computer.

Tip: IBM Informix products support IBM CCSID code-set numbers, a system of 16-bit numbers that uniquely identify the coded graphic character representations. For more information, see Appendix A, "Manage GLS files," on page A-1.

- One language can have several code sets. Each might represent a subset of the language.

  For example, the code sets **ccdc** and **big5** are both internal representations of a subset of the Chinese language. These subsets, however, include different numbers of Chinese characters.

Important: GLS fully supports the unified Chinese GB18030-2000 code set, including all characters in the Unicode Basic Multilingual Plane (BMP) and in the extended planes.

If a code-set conversion is required for data transfer from computer A to computer B, then it is also required for data transfer from computer B to computer A. In the client/server environment, the following situations might require code-set conversion:

- If the client locale and database locale specify different code sets, the client application performs code-set conversion so that the server computer is not loaded with this type of processing. For more information, see "Client application code-set conversion" on page 5-1.
- If the server locale and server-processing locale specify different code sets, the database server performs code-set conversion when it writes to and reads from operating-system files such as log files. For more information, see "Database server code-set conversion" on page 4-2.

In the following figure, the black dots indicate the two points in a client/server environment at which code-set conversion might occur.

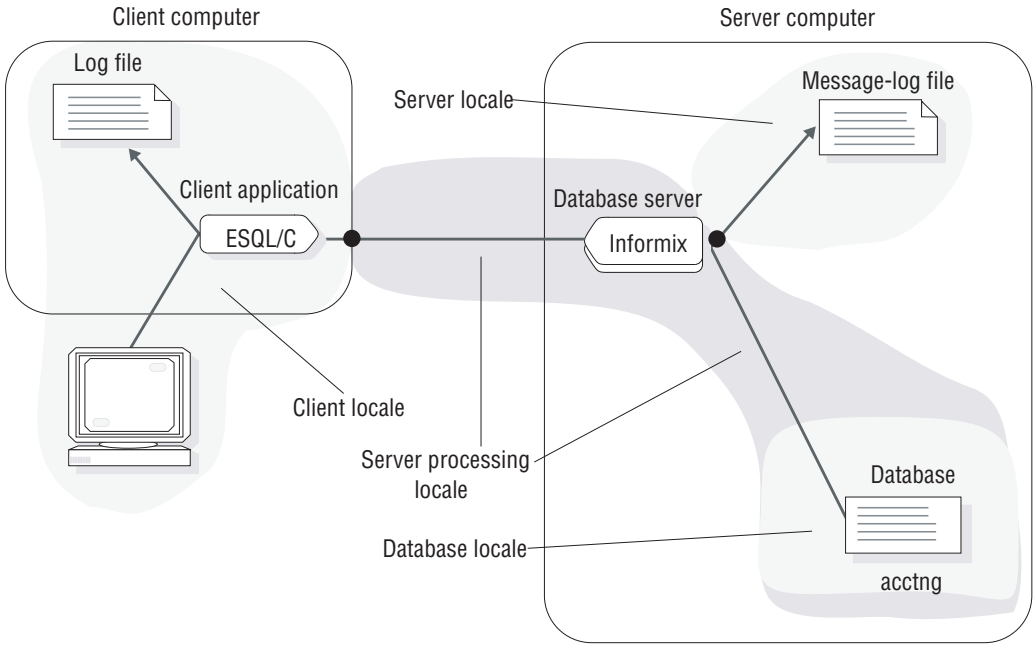


Figure 1-4. Points of GLS code-set conversion

In the example connection that the previous figure shows, the Informix ESQL/C client application performs code-set conversion on the data that it sends to and receives from the database server if the client and database code sets are

convertible. The IBM Informix database server also performs code-set conversion when it writes to a message-log file if the code sets of the server locale and server-processing locale are convertible.

Locate message files

IBM Informix products use GLS locales to locate product-specific message files. By default, IBM Informix products automatically search a subdirectory that is associated with the client locale for the product-specific message files.

The following table lists the subdirectory for each platform.

Platform	Directory
UNIX	`$INFORMIXDIR/msg/lg_tr/code_set`
Windows	`%INFORMIXDIR%\msg\lg_tr\code_set`

In this path, *lg* is the language and *tr* is the territory, from the name of the client locale, and *code_set* is the condensed form of the code-set name. For more information about condensed code-set names, see "Locale-file subdirectories" on page A-6.

IBM Informix products use a precedence of environment variables to locate product-specific message files. The **DBLANG** environment variable lets you override the client locale for the location of message files that IBM Informix products use. You might use **DBLANG** to specify a directory where the message files are for each locale that your environment supports.

Customize end-user formats

You can set environment variables to override end-user formats in the client locale.

You can override the following end-user formats in the client locale:

- End-user format of date and time (DATE, DATETIME) values
- End-user format of monetary (MONEY) values

This section explains how to customize these end-user formats. For an introduction to end-user formats, see "End-user formats" on page 1-13.

Customize date and time end-user formats

The GLS locales define end-user formats for dates and times, which you do not usually need to change, but customization is available.

You can customize end-user formats for DATE and DATETIME values (for example, 10-27-97 for the date 10/27/97) with the following environment variables.

Environment variable	Description
GL_DATE	Supports extended format strings for international formats in date end-user formats.
GL_DATETIME	Supports extended format strings for international formats in time end-user formats.
DBDATE	Specifies a date end-user format. (Supported for compatibility with earlier versions.)

Environment variable	Description
DBTIME	Specifies a time end-user format for certain embedded-language (ESQL) library functions. (Supported for compatibility with earlier versions.)

A date or time end-user format string specifies a format for the manipulation of internal DATE or DATETIME values as a literal string.

Tip: When you set these environment variables, you do not affect the internal format of the DATE and DATETIME values within a database.

The **GL_DATE** and **GL_DATETIME** environment variables support formatting directives that allow you to specify an end-user format. A formatting directive has the form %*x* (where *x* is one or more conversion characters).

Era-based date and time formats

The **GL_DATE** and **GL_DATETIME** environment variables provide support for alternative dates and times such as era-based (Asian) formats. These alternative formats support dates such as the Taiwanese Ming Guo year and the Japanese Imperial-era dates.

Tip: **DBDATE** and **DBTIME** can also provide some support for era-based dates.

To specify era-based formats for DATE and DATETIME values, use the `E` conversion modifier, as follows:

- For either **GL_DATE** or **GL_DATETIME**, `E` can show in several formatting directives.
  For a list of valid formatting conversions for eras, see "Alternative time formats" on page 2-17.
- For **DBDATE**, `E` can show in the format specification.

Date and time precedence

IBM Informix products use a precedence to determine the end-user format for an internal DATE value.

IBM Informix products use the following precedence:

1. **DBDATE**
2. **GL_DATE**
3. Information defined in the client locale (if **CLIENT_LOCALE** is set)
4. Default date format is `%m/%d/%iy` (if **DBDATE** and **GL_DATE** are not set, and no locale is specified)

IBM Informix products use the following precedence to determine the end-user format for an internal DATETIME value:

1. **DBDATE** and **DBTIME**
2. **GL_DATETIME**
3. Information that the client locale defines (**CLIENT_LOCALE**, if it is set)
4. Default DATETIME format is `%iY-%m-%d %H:%M:%S` (if **CLIENT_LOCALE**, **DBTIME** and **GL_DATETIME** are not set)

For more information about these formatting directives, see "The GL_DATE environment variable" on page 2-10 and "The GL_DATETIME environment variable" on page 2-15.

Customize monetary values

The GLS locales contain end-user formats, which you do not usually need to change. You can set the **DBMONEY** environment variable, however, to customize the appearance of the currency notation.

For information about the **DBMONEY** environment variable, see the *IBM Informix Guide to SQL: Reference*.

A monetary end-user format string specifies a format for the manipulation of internal DECIMAL, FLOAT, and MONEY values as monetary literal strings. IBM Informix products use the following precedence to determine the end-user format for a MONEY value:

1. **DBMONEY**
2. Information that the client locale defines.

   CLIENT_LOCALE identifies the client locale; if it is not set, the client locale is the default locale.
3. Default currency notation = $,.

   If **DBMONEY** is not set, and no locale is specified, the currency symbol is the dollar sign, the thousands separator is the comma, and the decimal separator is the period.

Chapter 2. GLS environment variables

IBM Informix products establish the client, database, and server locales with information from GLS-related environment variables and from data that is stored in the database.

These topics provide descriptions of the GLS-related environment variables. For more information about environment variables, see the *IBM Informix Guide to SQL: Reference*.

Set and retrieve environment variables

The GLS feature lets you use the diacritics, collating sequence, and monetary, date, and number conventions of the language that you select when you create databases.

No configuration parameters exist for GLS, but you must set the appropriate environment variables.

With IBM Informix ESQL/C, you can use the C **putenv()** function to modify, create, and delete environment variables, and the C **getenv()** function to retrieve the values of environment variables from the operating-system environment. For details, see the *IBM Informix ESQL/C Programmer's Manual*.

On UNIX platforms, set environment variables with the appropriate shell command (such as **setenv** for the C shell). For more information, see your UNIX documentation.

On Windows, set environment variables in the **InetLogin** structure or use the **Setnet32** utility to set environment variables in the `registry` file. For more information about **InetLogin**, see the Microsoft Windows documentation for your SQL API. For more information about **Setnet32**, see your *IBM Informix Installation Guide*.

Important: If you use **ifx_putenv()**, the application must set all environment variables before it calls any other IBM Informix library routine to avoid initializing the GLS routines and freezing the values of certain locale and formatting environment variables.

GLS-related environment variables

These topics list the GLS-related environment variables that you can set for IBM Informix database servers and SQL API products.

Important: Some previous releases of IBM Informix supported the **GL_PATH** environment variable. For all current versions of Informix, however, if you set **GL_PATH** before you initialize the database server (or any SUID/SGID programs provided by Informix) you get an error and its value is ignored.

© Copyright IBM Corp. 1996, 2013

The CC8BITLEVEL environment variable

The **CC8BITLEVEL** environment variable determines the type of processing that the IBM Informix ESQL/C filter, **esqlmf**, performs on non-ASCII (8-bit and multibyte) characters.

See also "Generate non-ASCII file names" on page 6-3.

Element	Description
0	The **esqlmf** filter converts all non-ASCII characters in literal strings and comments to octal constants (for C compilers that do not support these uses of non-ASCII characters).
1	The **esqlmf** filter converts non-ASCII characters in literal strings to octal constants but allows them in comments (some C compilers do support non-ASCII characters in comments).
2	The **esqlmf** filter allows non-ASCII characters in literal strings and ensures that all the bytes in the non-ASCII characters have the eighth bit set (for C compilers with this requirement).
3	The **esqlmf** filter does not filter non-ASCII characters (for C compilers that support multibyte characters in literal strings and comments).

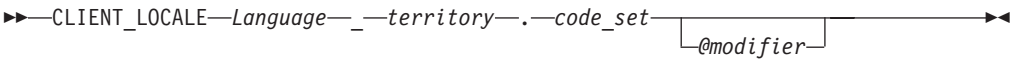
To start **esqlmf** each time that you process an Informix ESQL/C file with the **esql** command, set the **ESQLMF** environment variable to 1. If you do not set **CC8BITLEVEL**, the **esql** command assumes a value for **CC8BITLEVEL** of 0.

Important: For **ESQLMF** to take effect, do not set **CC8BITLEVEL** to 3.

The CLIENT_LOCALE environment variable

The **CLIENT_LOCALE** environment variable specifies the *client locale*, which the client application uses in read and write operations, end-user formats, and processing ESQL statements.

See also "The client locale" on page 1-18.



Element	Description
code_set	Name of the code set that the locale supports.
language	Two-character name that represents the language for a specific locale.
modifier	Optional locale modifier that has a maximum of four alphanumeric characters.

territory
Two-character name that represents the cultural conventions. For example, *territory* might specify the Swiss version of the French, German, or Italian language.

The *modifier* specification modifies the cultural-convention settings that the *language* and *territory* settings imply. The *modifier* usually indicates a special localized collating order that the locale supports.

An example nondefault client locale for a French-Canadian locale follows:

```
CLIENT_LOCALE fr_ca.8859-1
```

You can use the **glfiles** utility to generate a list of the GLS locales that are available on your UNIX system. For more information, see "The glfiles utility (UNIX)" on page A-12.

If you do not set **CLIENT_LOCALE**, the client application uses the default locale, U.S. English, as the client locale.

Changes to **CLIENT_LOCALE** also enter in the Windows `registry` database under HKEY_LOCAL_MACHINE.

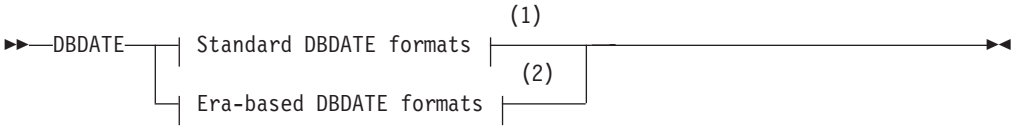
The DBDATE environment variable

The **DBDATE** environment variable specifies the end-user formats of values in DATE columns.

IBM Informix products support the **DBDATE** environment variable for compatibility with earlier products. It is recommend that you use the **GL_DATE** environment variable for new applications.

For information about end-user formats, see "End-user formats" on page 1-13.

Important: DBDATE is evaluated at system initialization time. If it is invalid, the system initialization fails.



Notes:

1 See *IBM Informix Guide to SQL: Reference*

2 See "DBDATE extensions" on page 6-7

Important: DBDATE variable takes precedence over the **GL_DATE** environment variable and over the default DATE formats that **CLIENT_LOCALE** specifies.

Related reference:

Environment variable changes by version (Migration Guide)

The DBLANG environment variable

The **DBLANG** environment variable specifies the subdirectory of **INFORMIXDIR** that contains the customized, language-specific message files that an IBM Informix product uses.

Element
Description

relative_path
Subdirectory of the IBM Informix installation directory (which **INFORMIXDIR** specifies)

full_path
Full path name of the directory that contains the compiled message files

locale_name
Name of a GLS locale that has the format *lg_tr.code_set*, where *lg* is a two-character name that represents the language for a specific locale, *tr* is a two-character name that represents the cultural conventions, and *code_set* is the name of the code set that the locale supports

IBM Informix products locate product-specific message files in the following order:

1. If **DBLANG** is set to a *full_path*: the directory that *full_name* indicates
2. If **DBLANG** is set to a *relative_path*:
   a. In `$INFORMIXDIR/msg/$DBLANG` on UNIX or `%INFORMIXDIR%\msg\%DBLANG%` on Windows
   b. In `$INFORMIXDIR/$DBLANG` on UNIX or `%INFORMIXDIR%\%DBLANG%` on Windows
3. If **DBLANG** is set to a *locale_name*: the `msg` subdirectory for the locale in `$INFORMIXDIR/msg/`*`lg_tr/code_set`* on UNIX systems or `%INFORMIXDIR%\msg\`*`lg_tr\code_set`* on Windows, where *lg* is the language, *tr* is the territory, and *code_set* is the code set in *locale_name*.

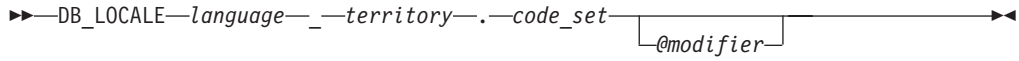
   The value of **DBLANG** does not affect the messages that the database server writes to its message log. The database server obtains the locale for these messages from the **SERVER_LOCALE** environment variable.
4. If **DBLANG** is not set: the `msg` subdirectory for the locale in `$INFORMIXDIR/msg/`*`lg_tr/code_set`* on UNIX systems or `%INFORMIXDIR%\msg\`*`lg_tr\code_set`* on Windows, where *lg* is the language and *tr* is the territory from the locale that is associated with the IBM Informix product, and *code_set* is the condensed name of the code set that the locale supports:
   - For IBM Informix client products: *lg* and *tr* are from the client locale (from **CLIENT_LOCALE**, if it is set)
   - For IBM Informix database server products: *lg* and *tr* are from the server locale (from **SERVER_LOCALE**, if it is set)
5. If **DBLANG**, **CLIENT_LOCALE**, and **LANG** are not set:
   a. In `$INFORMIXDIR/msg/en_us/0333` on UNIX systems or `%INFORMIXDIR%\msg\en_us\0333` on Windows, an internal message directory for the default locale
   b. In `$INFORMIXDIR/msg` on UNIX systems or `%INFORMIXDIR%\msg` on Windows, the default IBM Informix message directories

The compiled message files have the `.iem` file extension.

The DB_LOCALE environment variable

The **DB_LOCALE** environment variable specifies the *database locale*, which the database server uses to process locale-sensitive data.

See “The database locale” on page 1-19 and Appendix A, “Manage GLS files,” on page A-1.



Element
Description

code_set
Name of the code set that the locale supports.

language
Two-character name that represents the language for a specific locale.

modifier
Optional locale modifier that has a maximum of four alphanumeric characters.

territory
Two-character name that represents the cultural conventions. For example, *territory* might specify the Swiss version of the French, German, or Italian language.

The *modifier* specification modifies the cultural-convention settings that the *language* and *territory* settings imply. The *modifier* can indicate a localized collating order that the locale supports. For example, you can set *@modifier* to specify dictionary or telephone-book collation order.

An example nondefault database locale for a French-Canadian locale follows:

```
DB_LOCALE fr_ca.8859-1
```

The **glfiles** utility can generate a list of the GLS locales available on your UNIX system. For more information, see “The glfiles utility (UNIX)” on page A-12.

The SET COLLATION statement can specify for the current session a localized collation different from the COLLATION setting of the **DB_LOCALE** locale. This can affect sorting operations on NCHAR and NVARCHAR data values.

If you do not set **DB_LOCALE** on the client computer, client applications assume that the database locale has the value of the **CLIENT_LOCALE** environment variable. The client application, however, does not send this default value to the database server when it requests a connection.

Changes to **DB_LOCALE** also enter in the Windows `registry` database under HKEY_LOCAL_MACHINE.

The DBMONEY environment variable

The **DBMONEY** environment variable specifies the end-user formats for values in MONEY columns.

See also “End-user formats” on page 1-13.

Element
Description

front Specifies a currency symbol that is displayed before the monetary value.

back Specifies a currency symbol that is displayed after the value.

, (comma), . (period)
Monetary decimal separator. When you specify the comma or the period, you implicitly assign the other symbol to the thousands separator.

With this environment variable, you can specify the currency notation:

- The currency symbol that shows before or after the monetary value.
- The monetary decimal separator, which separates the part of the monetary value from the fractional part.

For example, suppose that you use ' DM,' as the **DBMONEY** setting. This **DBMONEY** setting specifies the following currency notation:

- The currency symbol, DM, shows before a monetary value.
- The decimal separator is a comma.
- The thousands separator is (implicitly) a period.

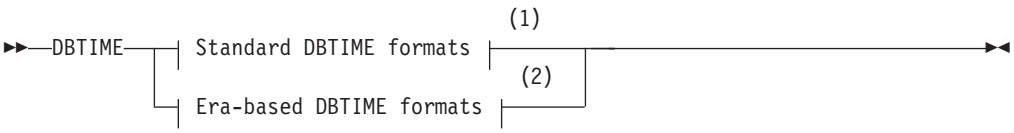
The *front* or *back* symbol can be non-ASCII character if your client locale supports a code set that defines the non-ASCII character. Any symbol that is not a letter must be enclosed within quotation marks. Period or comma are not valid *front* or *back* symbols. In the default locale, the dollar ($) sign is the default *front* currency symbol, period (.) is the default decimal separator, and comma (,) is the default thousands separator. The **DBMONEY** setting takes precedence over any notation defined by the MONETARY category of the locale. See also "Customize monetary values" on page 1-34.

Most GLS locales for European languages can support code sets that include the euro symbol for monetary values.

The DBTIME environment variable (ESQL/C)

The **DBTIME** environment variable specifies the end-user formats of values in DATETIME columns for SQL API routines.

IBM Informix products support **DBTIME** for compatibility with earlier products. It is recommended that you use the **GL_DATETIME** environment variable for new applications. See also "End-user formats" on page 1-13.



Notes:

1 See *IBM Informix Guide to SQL: Reference*

2 See "DBTIME support" on page 6-10

Tip: **DBTIME** affects only certain formatting routines in the ESQL/C function libraries. See "DATETIME-format functions" on page 6-9.

The ESQLMF environment variable

The **ESQLMF** environment variable can have the values 1 or 0.

Element
Description

0 The **esql** command compiles source code whose non-ASCII characters have already been converted.

1 The **esql** command calls **esqlmf** to filter multibyte characters when preprocessing an ESQL/C source file.

The **ESQLMF** environment variable indicates whether the **esql** command automatically starts the Informix ESQL/C multibyte filter, **esqlmf**.

The value of the **CC8BITLEVEL** environment variable determines the type of filtering that **esqlmf** performs. For information about **esqlmf**, see "Generate non-ASCII file names" on page 6-3.

Important: For **ESQLMF** to take effect, **CC8BITLEVEL** must not be set to 3.

If you want to compile existing source code whose non-ASCII characters have already been converted, either set **ESQLMF** to 0 or do not set it. In either case, **esql** does not start **esqlmf**.

The GLS8BITFSYS environment variable

Use the **GLS8BITFSYS** environment variable to tell IBM Informix products (such as the Informix ESQL/C processor) whether the operating system is 8-bit clean.

This setting determines whether an IBM Informix product can use non-ASCII characters in the file name of an operating-system file that it generates.

Element
Description

0 IBM Informix products assume that the operating system is not 8-bit clean and generate file names with 7-bit ASCII characters only.

1 IBM Informix products assume that the operating system is 8-bit clean and can use non-ASCII characters (8-bit or multibyte characters) in the file name of an operating-system file that it generates.

If you include non-ASCII characters in a file name that you specify within a client application, you must ensure that the code set of the server-processing locale supports these non-ASCII characters. If you do not set **GLS8BITFSYS**, IBM Informix database servers behave as if **GLS8BITFSYS** is set to 1.

For example, create a database that is called A1A2B1B2, where A1A2 and B1B2 are multibyte characters, with the following SQL statement:

```
CREATE DATABASE A1A2B1B2
```

If **GLS8BITFSYS** is 1 (or is not set) on the server computer, the database server assumes that the operating system is 8-bit clean, and it generates a database directory, `A1A2B1B2.dbs`.

If **GLS8BITFSYS** is set to 0 on the server computer and you include non-ASCII characters in the file name, the IBM Informix product uses an internal algorithm to convert these non-ASCII characters to ASCII characters. The file names that result are 7-bit clean.

File names with invalid byte sequences generate errors when they are used with GLS-based products.

Only some database utilities, such as **dbexport**, and the compilers for IBM Informix ESQL/C products use **GLS8BITFSYS** on the client computer to create and use files. For example, suppose you compile an Informix ESQL/C source file that is called `A1A2B1B2.ec`, where `A1A2` and `B1B2` are multibyte characters. If **GLS8BITFSYS** is set to 1 (or is not set) on the client computer, the Informix ESQL/C processor generates an intermediate C file that is called `A1A2B1B2.c`. For a list of Informix ESQL/C files that check **GLS8BITFSYS**, see "Handle non-ASCII characters" on page 6-1.

Restrictions on non-ASCII file names

If your locale supports a code set with non-ASCII characters, restrictions apply to file names for operating-system files that IBM Informix products generate.

Before you or an IBM Informix product creates a file and assigns a file name, consider the following questions:

- Does your operating system support non-ASCII file names?
- Does the IBM Informix product accept non-ASCII file names?

Make sure that your operating system is 8-bit clean:

To support non-ASCII characters in file names, your operating system must be *8-bit clean*.

An operating system is 8-bit clean if it reads the eighth bit as part of the code value. In other words, the operating system must not ignore or make its own interpretation of the value of the eighth bit.

Consult your operating-system manual or system administrator to determine whether your operating system is 8-bit clean before you decide to use a nondefault locale that contains non-ASCII characters in file names that IBM Informix products use and generate.

Make sure that your product supports the same code set:

After an IBM Informix product has generated an operating-system file whose file name has non-ASCII characters, it has written that file name and the file contents in a particular code set.

Whenever an IBM Informix product or client application must access that file, you must ensure that the product uses a server-processing locale that supports that same code set.

The server code set:

When the database server creates a file whose file name contains non-ASCII characters, the server locale must support these characters.

Before you start a database server, you must set the **SERVER_LOCALE** environment variable to the name of a locale whose code set contains these non-ASCII characters.

For example, suppose you want a message log with the UNIX path /A1A2B1B2/C1C2D1D2, where A1A2, B1B2, C1C2, and D1D2 are multibyte characters in the Japanese SJIS code set. To enable the database server to create this message-log file on its computer:

1. Modify the MSGPATH parameter in the ONCONFIG file.

   For UNIX:

   ```
   MSGPATH /A1A2B1B2/C1C2D1D2
   # multibyte message-log filename
   ```

   For Windows:

   ```
   MSGPATH \A1A2B1B2\C1C2D1D2
   # multibyte message-log filename
   ```

2. Set the **SERVER_LOCALE** environment variable on the server computer to the Japanese SJIS locale, **ja_jp.sjis**.
3. Start the database server with the **oninit** utility.

When the database server initializes, it assumes that the operating system is 8-bit clean and creates the /A1A2B1B2/C1C2D1D2 message log on UNIX, or the \A1A2B1B2\C1C2D1D2 file on Windows.

The client code set:

When an Informix ESQL/C processor creates a file whose file name has non-ASCII characters, the client locale must support these non-ASCII characters.

Before you start an IBM Informix database server, you must ensure that the code set of the client locale (the client code set) contains these characters.

When you use a nondefault locale, you must set the **CLIENT_LOCALE** environment variable to the name of a locale whose code set contains these non-ASCII characters.

For example, suppose you want to process an Informix ESQL/C source file with the path /A1A2B1B2/C1C2D1D2, where A1A2, B1B2, C1C2, and D1D2 are multibyte characters in the Japanese SJIS code set. You must perform the following steps to enable the **esql** command to create the intermediate C source file on the client computer:

1. Set the **CLIENT_LOCALE** environment variable on the client computer to the Japanese SJIS locale, **ja_jp.sjis**.
2. Process the Informix ESQL/C source file with the **esql** command.

If the code sets that are associated with the file name and with the client locale do not match, a valid file name might contain invalid characters with respect to the client locale. The Informix ESQL/C processor rejects any file name that contains invalid characters and the following error message is displayed:

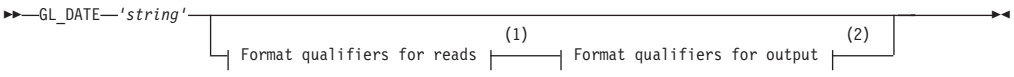
```
Illegal characters in filename.
```

The GL_DATE environment variable

The **GL_DATE** environment variable specifies end-user formats of values for DATE columns.

For information about end-user formats, see "End-user formats" on page 1-13.

Important: **GL_DATE** is evaluated when it is used, rather than when it is set. If it is invalid, the operation that called it fails.



Notes:

1 See "Field specification for reading a DATE value" on page 2-13

2 See "Field specification for displaying a DATE value" on page 2-14

Element
Description

string Formatting directives that specify the end-user format for **GL_DATE** values. You can use any formatting directive that formats dates.

An end-user format in **GL_DATE** can contain the following characters:

- One or more white space characters, which the CTYPE category of the locale specifies.
- An ordinary character (other than the % symbol or a white-space character).
- A formatting directive, which is composed of the % symbol followed by a conversion character that specifies the required replacement.

The next list describes the formatting directives that are not based on era.

Formatting directives	Description
%a	Is replaced by the abbreviated weekday name as defined in the locale.
%A	Is replaced by the full weekday name as defined in the locale.
%b	Is replaced by the abbreviated month name as defined in the locale.
%B	Is replaced by the full month name as defined in the locale.
%C	Is replaced by the century number (the year divided by 100 and truncated to an integer) as an integer (00 through 99).
%d	Is replaced by the day of the month as an integer (01 through 31). A single digit is preceded by a zero (0).
%D	Is the same as the %m/%d/%y format.
%e	Is replaced by the day of the month as a number (1 through 31). A single digit is preceded by a space.
%h	Is the same as the %b formatting directive.
%iy	Is replaced by the year as a two-digit number (00 - 99) for both reading and printing. It is the formatting directive specific to IBM Informix for %y.
%iY	Is replaced by the year as a four-digit number (0000 - 9999) for both reading and printing. It is the formatting directive specific to IBM Informix for %Y.
%m	Is replaced by the month as a number (01 through 12).

Formatting directives	Description
%n	Is replaced by a newline character.
%t	Is replaced by the TAB character.
%w	Is replaced by the weekday as a number (0 through 6); 0 represents the locale equivalent of Sunday.
%x	Is replaced by a special date representation that the locale defines.
%y	Requires that the year is a two-digit number (00 through 99) for both reading and printing.
%Y	Requires that the year is a four-digit number (0000 through 9999) for both reading and printing.
%%	Is replaced by % (to allow % in the format string).

White space or other nonalphanumeric characters must show between any two formatting directives. For example, if you use a U.S. English locale, you might want to format an internal DATE value for 03/05/1997 in the ASCII string format that the following example shows:

```
Mar 05, 1997 (Wednesday)
```

To do so, set the **GL_DATE** environment variable as follows:

```
%b %d, %Y (%A)
```

If a **GL_DATE** format does not correspond to any of the valid formatting directives, the behavior of the IBM Informix product when it tries to format is undefined.

Important: The setting of the **DBDATE** variable takes precedence over that of the **GL_DATE** environment variable and over the default DATE formats that **CLIENT_LOCALE** specifies.

The year formatting directives

You can use these formatting directives in the end-user format of the **GL_DATE** environment variable to format the year of a date string: %y, %iy, %Y, and %iY.

The %iy and %iY formatting directives provide compatibility with the Y2 and Y4 year specifiers of the **DBDATE** environment variable.

For information about end-user formats, see "End-user formats" on page 1-13.

When an IBM Informix product uses an end-user format to *print* an internal date value as a string, the %iy directive performs the same task as % y, and %iY directive performs the same task as %Y. To print a year with one of these formatting directives, an IBM Informix product performs the following actions:

- The %iy and %y formatting directives both print the year of an internal date value as a two-digit decade.

  For example, when you set **GL_DATE** to '%y %m %d' or '%iy %m %d', an internal date for March 5, 1997 formats to '97 03 05'.
- The %iY and %Y formatting directives both print the year of an internal date value as a four-digit year.

  For example, when you set **GL_DATE** to '%Y %m %d' or '%iY %m %d', the internal date for March 5, 1997 formats to '1997 03 05'.

When an IBM Informix product uses an end-user format to *read* a date, the %iy formatting directive performs differently from %y and the %iY formatting directive performs differently %Y. The following table summarizes how the year formatting directives behave when an IBM Informix product uses them to read date strings.

Table 2-1. GL_DATE formats with sample date strings

GL_DATE format	'1994 03 06' date string to read	'94 03 06' date string to read
%y %m %d	Error	Internal date for 1994 03 06
%iy %m %d	Internal date for 1994 03 06	Internal date for 1994 03 06
%Y %m %d	Internal date for 1994 03 06	Internal date for 0094 03 06
%iY %m %d	Internal date for 1994 03 06	Internal date for 1994 03 06

In a read of a date string, the %iy and %y formatting directives both prefix the first two digits of the current year to expand any one-digit or two-digit year. You can set the **DBCENTURY** environment variable to change this default.

Alternative date formats

To support alternative date formats in an end-user format, **GL_DATE** accepts the *conversion modifiers*.

These conversion modifiers are:

- E indicates use of an alternative era format, which the locale defines.
- O (the letter O) indicates use of locale-defined alternative digits.

These date-formatting directives can support conversion modifiers.

Date format	Description
%EC	Accepts either the full or the abbreviated era name for reading; for printing, %EC is replaced by the full name of the base year (period) of the era that the locale defines (same as %C if locale does not define an era).
%Eg	Accepts the full or the abbreviated era name for reading. For printing, %Eg is replaced by the abbreviated name of the base year (period) of the era that the locale defines (same as %C if locale does not define an era).
%Ex	Is replaced by a special date representation for an era that the locale defines (same as %x if locale does not define an era).
%Ey	Is replaced by the offset from %EC of the era that the locale defines. This date is the era year only (same as %y if locale does not define an era).
%EY	Is replaced by the full era year, which the locale defines (same as %Y if locale does not define an era).
%Od	Is replaced by the day of the month in the alternative digits that the locale defines (same as %d if locale does not define alternative digits).
%Oe	Is the same as %Od (or %e if locale does not define alternative digits).
%Om	Is replaced by the month in the alternative digits that the locale defines (same as %m if locale does not define alternative digits).
%Ow	Is replaced by the weekday as a single-digit number (0 through 6) in the alternative digits that the locale defines (same as %w if locale does not define alternative digits). The equivalent of zero (0) represents the locale equivalent of Sunday.

Date format	Description
%Oy	Is replaced by the year as a two-digit number (00 through 99) in the alternative digits that the locale defines (same as %y if locale does not define alternative digits). For information about how to format a year value, see the description of %y.
%OY	Is the same as %EY (or %Y if locale does not define alternative digits).

The TIME category of the locale defines the following era information:

- The full and abbreviated names for an era
- A representation for the era (which the %Ex directive uses)

The NUMERIC category of the locale defines the alternative digits for a locale (which the %O*x* formatting directives use).

Optional date format qualifiers

You can specify *optional format qualifiers* immediately after the % symbol of the formatting directive.

A date format qualifier defines a field specification for the date in read or print operations. The following sections describe what a field specification means for the read and print operations. For information about end-user formats, see "End-user formats" on page 1-13.

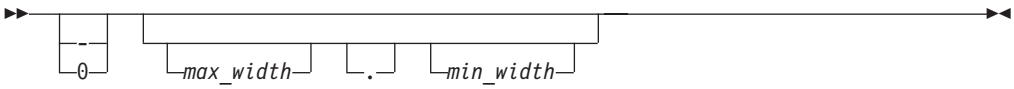
Tip: The **GL_DATETIME** environment variable accepts these date format qualifiers in addition to those qualifiers that "Optional time format qualifiers" on page 2-18 lists.

Field specification for reading a DATE value:

When an IBM Informix product uses an end-user format to read a date string, the field specification defines the number of characters to expect as input.

This field specification has the following syntax.

Format qualifiers for reads



Element
Description

- (minus sign)
Field value is left-aligned and begins with a digit; this value can include trailing spaces.

0 (zero)
Field value is right-aligned; any zeros on the left are pad characters that are not significant.

max_width
Integer that indicates the maximum number of characters to read.

min_width
Integer that indicates the minimum number of characters to read.

The first character of the field specification indicates whether to assume that the field value is justified or padded. If the first character is not a minus sign or a zero, the IBM Informix product assumes that the field value is right-aligned and any spaces on the left are pad characters.

If the field value begins with a zero, however, it cannot include pad characters.

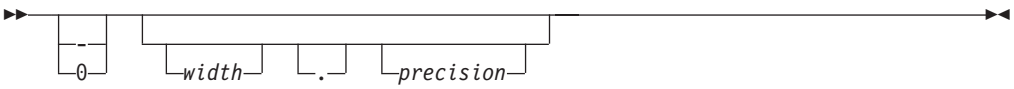
An IBM Informix product ignores the field specification if the field value is not a numeric value.

Field specification for displaying a DATE value:

When an IBM Informix product uses an end-user format to print a date string, the field specification defines the number of characters to print as output.

The syntax for the field specification is as follows.

Format qualifiers for output



Element
Description

- (minus sign)
Field value is left-aligned and begins with a digit; value can include trailing blank spaces.

0 (zero)
Field value is right-aligned; any zeros on the left are pad characters; they are not significant.

width Integer that indicates a minimum field width for the printed value.

precision
Integer that indicates the precision to use for the field value.

The meaning of the *precision* value depends on the formatting directive with which it is used, as the following table shows.

Formatting Directives	Description
%C, %d, %e, %Ey, %iy, %iY,%m, %w, %y, %Y	Value of *precision* specifies the minimum number of digits to print. If a value supplies fewer digits than *precision* specifies, an IBM Informix product pads the value with leading zeros. The %d, %Ey, %iy, %m, %w, and %y formatting directives have a default precision of 2. The %Y directive has no precision default; year 0001 would be formatted as 1 rather than as 0001.
%a, %A, %b, %B, %EC, %Eg, %h	Value of *precision* specifies the maximum number of characters to print. If a value supplies more characters than *precision* specifies, an IBM Informix product truncates the value.

Formatting Directives	Description
%D	Values of *width* and *precision* affect each element of these formatting directives. For example, the field specification %6.4D causes a DATE value to be displayed as if the format were: %6.4m/%6.4d/%6.4y where no fewer than four (but no more than six) characters represented the month, day, and year values, in that order, with "/" as the separator.
%O*x*	For formatting directives that include the **O** modifier (alternative digits), the value of *precision* is still the minimum number of digits to print. The *width* value defines the format width rather than the actual number of digits.
%Ex, %EY, %n, %t, %x, %%	Values of *width* and *precision* have no effect on these formatting directives.

For example, the following formatting directive displays the month as an integer with a maximum field width of 4: `%4m`

The following formatting directive displays the day of the month as an integer with a minimum field width of 3 and a maximum field width of 4: `%4.3d`

The GL_DATETIME environment variable

The **GL_DATETIME** environment variable specifies the end-user format of values in DATETIME columns.

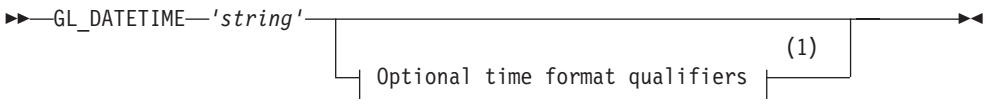
Like **DBDATE**, **DBTIME**, and **GL_DATE**, the **GL_DATETIME** setting controls only the character-string representation of data values. It has no effect on the internal storage format of DATETIME columns. For more information about end-user formats, see "End-user formats" on page 1-13.

Important:

In a database where **GL_DATETIME** has a nondefault setting, you cannot process localized DATETIME values correctly in some operations that load or unload data unless the **USE_DTENV** environment variable is set to 1. For more information about this dependency of **GL_DATETIME** on the **USE_DTENV** setting, see "The USE_DTENV environment variable" on page 2-20.

The setting of a **GL_DATETIME** format can contain the following characters:

- One or more white space characters, which the CTYPE category of the locale specifies
- An ordinary character (other than the % symbol or a white-space character)
- A formatting directive, which is composed of the % symbol, immediately followed by one or more conversion characters that specifies the required replacement.



Notes:

1 See "Optional time format qualifiers" on page 2-18

Element	Description
string	Contains the formatting directives that specify the end-user format for DATETIME values. You can use any formatting directive that formats dates or points in time. (For a list of additional formatting directives for year, month, and day values that are also valid in the **GL_DATETIME** setting, see "The GL_DATE environment variable" on page 2-10.)

The following list describes the DATETIME formatting directives that are not based on era.

Formatting directives	Description
%c	Is replaced by a special DATETIME representation that the locale defines.
%F*n*	Is replaced by the value of the fraction of a second, with precision that is specified by the unsigned integer *n*. The default value of *n* is 2; the range of *n* is 0 ≤ *n* ≤ 5. This value overrides any width or precision between the % and F character. For more information, see "Optional time format qualifiers" on page 2-18.
%H	Is replaced by the hour as an integer (00 through 23) for a 24-hour clock format.
%I	Is replaced by the hour as an integer (00 through 11) for a 12-hour clock format.
%M	Is replaced by the minute as an integer (00 through 59).
%p	Is replaced by the A.M. or P.M. equivalent, as defined in the locale.
%r	Is replaced by the commonly used time representation for a 12-hour clock format, including the A.M. or P.M. equivalent, as defined in the locale.
%R	Is replaced by the time in 24-hour notation (%H:%M).
%S	Is replaced by the second as an integer (00 through 61). The second can be up to 61 instead of 59 to allow for the occasional leap second and double leap second.
%T	Is replaced by the time in the %H:%M:%S format.
%X	Is replaced by the commonly used time representation as defined in the locale.
%%	Is replaced by % (to allow a literal % character in the format string).

Important: Any separator character between the %S and %F directives for DATETIME user formats must be explicitly defined. In IBM Informix release versions earlier than 12.10.xC1 and 11.70.xC8, the %F directive inserted by default the ASCII 46 character (.) between the SECOND and FRACTION field values. In this release, however, there is no default separator. Consecutive %S%F directives concatenate the digits representing the integer and fractional parts of the seconds value in the end-user format.

Within the *format* string, white space or other nonalphanumeric characters must show between any two formatting directives. Any other characters in the **GL_DATETIME** setting that were not listed in the table above or in the **GL_DATE** environment variable description as formatting directives are interpreted as literal characters. If a **GL_DATETIME** format does not correspond to any of the valid formatting directives, the behavior of the IBM Informix product when it tries to format DATETIME values is undefined.

"The GL_DATE environment variable" on page 2-10 describes additional formatting directives that you can also include in the **GL_DATETIME** setting to specify the end-user format of DATETIME values:

```
%a, %A, %b, %B, %C, %d, %D, %e, %h, %iy, %iY, %m, %n, %t,
%w, %x, %y, %Y
```

For example, if you use an U.S. English locale, you might want to format an internal DATETIME YEAR TO SECOND value to the ASCII string format that the following example shows:

```
Mar 21, 2013 at 16 h 30 m 28 s
```

To do so, set the **GL_DATETIME** environment variable as the following line shows:

```
%b %d, %Y at %H h %M m %S s
```

Important: The setting of **GL_DATETIME** affects the behavior of certain Informix ESQL/C library functions if the **DBTIME** environment variable is not set. For information about how these library functions are affected, see "DATETIME-format functions" on page 6-9. The setting of **DBETIME** takes precedence over the setting of **GL_DATETIME**.

Related reference:

Environment variable changes by version (Migration Guide)

Alternative time formats

To support alternative time formats in an end-user format, **GL_DATE** accepts the *conversion modifiers*.

The conversion modifiers are:

- `E` indicates use of an alternative era format, which the locale defines.
- `O` (the letter O) indicates use of alternative digits, which the locale also defines.

The following table shows time-formatting directives that support conversion modifiers.

Alternative time format	Description
%Ec	Is replaced by a special date/time representation for the era that the locale defines. It is the same as %c if the locale does not define an era.
%EX	Is replaced by a special time representation for the era that the locale defines. It is the same as %X if the locale does not define an era.
%OH	Is replaced by the hour in the alternative digits that the locale defines (24-hour clock). It is the same as %H if the locale does not define alternative digits).
%OI	Is replaced by the hour in the alternative digits that the locale defines (12-hour clock). It is the same as %I if the locale does not define alternative digits).
%OM	Is replaced by the minute with the alternative digits that the locale defines. It is the same as %M if the locale does not define alternative digits.
%OS	Is replaced by the second with the alternative digits that the locale defines. It is the same as %S if the locale does not define alternative digits.

The TIME category of the locale defines the following era information:

- The full and abbreviated names for an era

- A special date representation for the era (which the %Ex formatting directive uses)
- A special time representation for the era (which the %EX formatting directive uses)
- A special date/time representation for the era (which the %Ec formatting directive uses)

The NUMERIC category of the locale defines the alternative digits for a locale (which the %Ox formatting directives use).

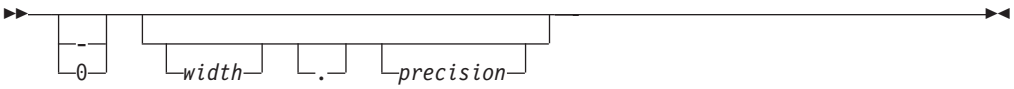
Optional time format qualifiers

You can specify the *optional format qualifiers* immediately after the % symbol of the formatting directive.

A time format qualifier defines a field specification for the time (or date and time) that the IBM Informix product reads or prints. This section describes what a field specification means for the print operation. For a description of what a field specification means for the read operation, see "Field specification for reading a DATE value" on page 2-13. For information about end-user formats, see "End-user formats" on page 1-13.

When an IBM Informix product uses an end-user format to print a string from an internal format, the field specification defines the number of characters to print as output. This field specification has the following syntax.

Optional time format qualifiers



Element
Description

- (minus sign)
IBM Informix product prints the field value as left-aligned and pads this value with spaces on the right.

0 (zero)
IBM Informix product prints the field value as right-aligned and pads this value with zeros on the left.

width Integer that indicates a minimum field width for the printed value.

precision
Integer that indicates the precision to use for the field value.

The first character of the field specification indicates whether to justify or pad the field value. If the first character is not a minus sign or a zero, an IBM Informix product prints the field value as *right-aligned* and pads this value with spaces on the left.

The meaning of the precision value depends on the particular formatting directive with which it is used, as the following table shows.

Formatting directives	Description
%F, %H, %I, %M, %S	Value of *precision* specifies the minimum number of digits to print. If a value supplies fewer digits than the *precision* specifies, an IBM Informix product pads the value with leading zeros. The %H, %M, and %S formatting directives have a default precision of 2.
%p	Value of *precision* specifies the maximum number of characters to print. If a value supplies more characters than the *precision* specifies, an IBM Informix product truncates the value.
%R, %T	Values of *width* and *precision* affect each element of these formatting directives. For example, the field specification %6.4R causes a DATETIME value to be displayed if the format were %6.4H:%6.4M. Here no fewer than four (but no more than six) characters represented the hour and the minute.
%F	Value of *precision* can follow this directive as an optional precision specification. This value must be 1 - 5. Otherwise, an IBM Informix product generates an error. This precision value overrides any *precision* value that you specify between the % symbol and the formatting directive.
%O*x*	For formatting directives that include the **O** modifier, value of *precision* is still the minimum number of digits to print. The *width* value defines the format width, rather than the actual number of digits.
%c, %Ec, %EX, %X	Values of *width* and *precision* have no effect on these formatting directives.

For example, the following formatting directive displays the minute as an integer with a maximum field width of 4: `%4M`

The following formatting directive displays the hour as an integer with a minimum field width of 3 and a maximum field width of 6: `%6.3H`

The specified format is applied to all displayed DATETIME values, regardless of their declared precision. For example, suppose that the setting of **GL_DATETIME** is `'%Y/%m/%d %H:%M:%S'` This setting would cause a value from a DATETIME YEAR TO SECOND column to be displayed as follows:

```
[2000/08/28 14:43:17]
```

If a program executed on August 28 of the year 2000, the same **GL_DATETIME** setting would also display a value from a DATETIME HOUR TO SECOND column as follows:

```
[2000/08/28 14:43:17]
```

When **GL_DATETIME** is set, every DATETIME value is displayed in the specified format, even if that format includes time units that were not included in the DATETIME qualifier when the data type was declared. Time units outside the declared precision are obtained from the system clock-calendar. To avoid unexpected results, you might prefer to set **GL_DATETIME** only for applications where the DATETIME data types that you display have the same precision as the **GL_DATETIME** setting.

Creation-time settings

Like **DBCENTURY**, **DBDATE**, and **GL_DATE**, the **GL_DATETIME** variable can affect how expressions that include literal time values are evaluated.

For some earlier releases, resetting environment variables can produce inconsistent behavior in check constraints, triggers, fragmentation expressions, UDRs, and other database objects whose definitions include time expressions. Objects created in this release use the environment variable settings that were in effect at the time when the object was created, rather than the settings at the time of execution (if these settings are different) to avoid inconsistency.

The USE_DTENV environment variable

In a database where the **GL_DATETIME** environment variable has a nondefault setting, you cannot process localized DATETIME values correctly in some operations that load or unload data unless **USE_DTENV** is enabled.

In a nondefault locale, if **GL_DATETIME** defines a nondefault DATETIME end-user format, the **USE_DTENV** environment variable must be set to 1 for DATETIME values to be processed correctly in the following contexts:

- **dbexport** utility
- **dbimport** utility
- LOAD statement of DB-Access
- UNLOAD statement of DB-Access
- DML operations on objects defined by the CREATE EXTERNAL TABLE statement.

For additional information about the **USE_DTENV** environment variable, see the *IBM Informix ESQL/C Programmer's Manual*.

The GL_USEGLU environment variable

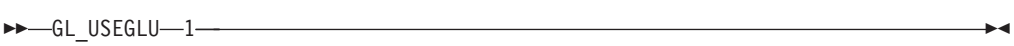
For IBM Informix to support versions of Unicode up to 4.1, the **GL_USEGLU** environment variable must be set to a value of 1 (one) in the server environment before the server is started or before the database is created. This setting initializes conversion routines that enable Unicode collation by the server in databases that use **UTF-8** character encoding, including the Chinese **GB18030-2000** code set. The conversion applies only to databases that were created with GL_USEGLU=1 already set.

A database that was created with **GL_USEGLU** set to 1 must be accessed with **GL_USEGLU** set to 1. For example, the SET COLLATION statement of SQL cannot enable localized collation for a nondefault Unicode locale, such as **sh_hr.utf8**, which supports the Serbo-Croatian language, unless **GL_USEGLU** was set to 1 when the server was started.

When you set the **GL_USEGLU** environment variable to 1, also set the STACKSIZE configuration parameter in the `onconfig` file to at least 64 KB.

Additionally, the GL_USEGLU setting must match between the source and target server during migration.

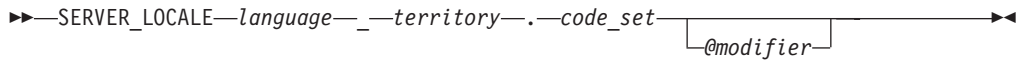
To enable Unicode collation by Java™ JDBC or ESQL/C client applications, and for other client APIs that require compilation, set **GL_USEGLU** to 1 in the client environment before the routine is compiled. For ESQL/C applications, an alternative to setting **GL_USEGLU** is to compile with the **-glu** flags of **esql** when linking the ESQL/C program, in order to access internal Unicode libraries.



The SERVER_LOCALE environment variable

The **SERVER_LOCALE** environment variable specifies the *server locale*, which the database server uses to perform read and write operations that involve operating-system files on the server computer.

For more information about the server locale, see "The server locale" on page 1-20 and "GLS support by IBM Informix database servers" on page 4-1.



Element
Description

code_set
Name of the code set that the locale supports

language
Two-character name that represents the language for a specific locale

modifier
Optional locale modifier that has a maximum of four alphanumeric characters.

territory
Two-character name that represents the cultural conventions. For example, *territory* might specify the Swiss version of the French, German, or Italian language.

The *modifier* specification modifies the cultural-convention settings that the *language* and *territory* settings imply. The *modifier* usually indicates a special type of localized collation that the locale supports. For example, you can set @*modifier* to specify dictionary or telephone-book collating order.

An example nondefault server locale for a French-Canadian locale follows:

```
SERVER_LOCALE fr_ca.8859-1
```

You can use the **glfiles** utility to generate a list of the GLS locales that are available on your UNIX system. For more information, see "The glfiles utility (UNIX)" on page A-12.

If you do not set **SERVER_LOCALE**, IBM Informix database servers use the default locale, U.S. English, as the server locale.

Changes to **SERVER_LOCALE** also enter in the Windows `registry` database under HKEY_LOCAL_MACHINE.

Chapter 3. SQL features

These topics explain how the GLS feature affects the IBM Informix implementation of SQL.

For more information about the IBM Informix implementation of SQL, see the *IBM Informix Guide to SQL: Syntax*, the *IBM Informix Guide to SQL: Reference*, the *IBM Informix Guide to SQL: Tutorial*, and the *IBM Informix Database Design and Implementation Guide*.

Name database objects

You must declare names for new database objects (and in some cases, for storage objects, such as dbspaces) when you use data definition language (DDL) statements such as CREATE TABLE, CREATE INDEX, and RENAME COLUMN.

These topics describe considerations for declaring names for database objects in a nondefault locale. In particular, this section explains which SQL identifiers and delimited identifiers accept non-ASCII characters.

Important: To use a nondefault locale, you must set the appropriate locale environment variables for IBM Informix products. For more information, see "Set a nondefault locale" on page 1-23.

Rules for identifiers

An SQL identifier is a string of letters, digits, and underscores that represents the name of a database object such as a table, column, index, or view.

A non-delimited SQL identifier must begin with a letter or underscore (_) symbol. Trailing characters in the identifier can be any combination of letters, digits, underscores, or dollar ($) signs. Delimited identifiers, however, can include any character in the code set of the database locale; see "Delimited identifiers" on page 3-4 for more information.

Declaring identifiers that are SQL keywords can cause syntactic ambiguity or unexpected results. For additional information, see the Identifier segment in the *IBM Informix Guide to SQL: Syntax*. See also "Non-ASCII characters in identifiers" and "Valid characters in identifiers" on page 3-5.

SQL identifiers can occupy up to 128 bytes on IBM Informix. When you declare identifiers, make sure not to exceed the size limit for your database server. For example, the following statement creates a synonym name of eight multibyte characters:

```
CREATE SYNONYM A1A2A3B1B2C1C2C3D1D2E1E2F1F2G1G2H1H2 FOR A1A2B1B2
```

Non-ASCII characters in identifiers

IBM Informix database servers support non-ASCII (wide, 8-bit, and multibyte) characters from the code set of the database locale in most SQL identifiers, such as the names of columns, connections, constraints, databases, indexes, roles, SPL routines, sequences, synonyms, tables, triggers, and views.

© Copyright IBM Corp. 1996, 2013

On IBM Informix, you can use non-ASCII characters (8-bit and multibyte characters) when you create or refer to any of these database server names:

- Chunk name
- Message-log file name
- Path name

The following restrictions affect the ability of the database server to generate file names that contain non-ASCII characters:

- The database server must know whether the operating system is 8-bit clean.
- The code set specified by the **DB_LOCALE** setting must support these non-ASCII characters.

In a database with a nondefault locale, whose code set supports multibyte (or other non-ASCII) characters, you can use those non-ASCII characters when you declare most SQL identifiers, as listed in the following table.

In the following table, the **Type of identifier** column lists various categories of objects that can have SQL identifiers or operating-system identifiers. The **SQL segment** column shows the segment that provides the syntax of the identifier in the *IBM Informix Guide to SQL: Syntax*. The **Example context** column lists an SQL statement that can declare or can reference the identifier.

Table 3-1. SQL identifiers that support non-ASCII characters

Type of identifier	SQL segment	Example context
Alias	Identifier	SELECT
Cast	Expression	CREATE CAST
Column name	Identifier	CREATE TABLE
Connection name	Quoted String	CONNECT (For more information, see "Specify quoted strings" on page 3-12.)
Constraint name	Database Object Name	CREATE TABLE
Cursor name	Identifier	DECLARE (For more information, see "Handle non-ASCII characters" on page 6-1.)
Database name	Database Object Name	CREATE DATABASE
Distinct data type name	Identifier, Data Type	CREATE DISTINCT
File name	None	LOAD
Function name	Database Object Name	CREATE FUNCTION
Host variable	None	FETCH (For more information, see "Handle non-ASCII characters" on page 6-1.)
Index name	Database Object Name	CREATE INDEX
Opaque data type name	Identifier, Data Type	CREATE OPAQUE TYPE
Operator-class name	Database Object Name	CREATE OPCLASS
Partition	Identifier	ALTER FRAGMENT
Routine name	Database Object Name	CREATE FUNCTION
Routine name	Database Object Name	CREATE PROCEDURE

Table 3-1. SQL identifiers that support non-ASCII characters (continued)

Type of identifier	SQL segment	Example context
Role name	Identifier	CREATE ROLE
Row data type	Identifier	CREATE ROW TYPE
Sequence name	Database Object Name	CREATE SEQUENCE
SQL Statement identifier	Identifier	PREPARE (For more information, see "Handle non-ASCII characters" on page 6-1.)
SPL routine name	Database Object Name	CREATE PROCEDURE
SPL routine variables	None (language-specific)	CREATE PROCEDURE FROM
Synonym	Database Object Name	CREATE SYNONYM
Table name	Database Object Name	CREATE TABLE
Trigger correlation name	Database Object Name	CREATE TRIGGER
Trigger name	Database Object Name	CREATE TRIGGER
View name	Database Object Name	CREATE VIEW

Qualifiers of SQL identifiers

A complete syntax can include other identifiers.

The **SQL segment** column in Table 3-1 on page 3-2 shows the segment in the *IBM Informix Guide to SQL: Syntax* that describes the syntax of the identifier. In many cases, the complete syntax can include other identifiers. For example, the Database Object Name segment shows that the syntax of an index name can also include a *database* name, a database *server* name, and an *owner* name and the unqualified name of the index.

Keep in mind that even if the simple, unqualified name of a database object accepts multibyte characters, other identifiers in the fully qualified name of that object, such as *database@server:owner.index*, can include multibyte characters only if they also appear in the previous table. In this example, the *database* qualifier within the fully qualified index name can include multibyte characters, but the identifier of the database *server* that qualifies the *index* name cannot include multibyte characters.

Owner names

The *owner name* is the identifier of the user (or of a pseudo-user, for an owner like **informix** that does not correspond to the login name of an actual user) who is associated with the creation of a database object.

The owner name qualifies the identifier of the database object, which the owner typically can modify or drop. A synonym for the term *owner name* is *authorization identifier*. Unlike SQL identifiers, an authorization identifier cannot be longer than 32 bytes.

The ANSI term for owner name is *schema name*. In an ANSI-compliant database, you must specify the owner name as a qualifier of the identifier of any database object that you do not own.

Non-ASCII characters are not valid in an owner name unless your operating system supports those characters in user names.

If your database server is on a UNIX system, the owner-name qualifier defaults to the UNIX login ID. Most versions of UNIX, however, do not support multibyte characters in UNIX login IDs.

Important: You specify multibyte characters in an owner name at your own risk. If a UNIX login ID is used to match the owner name, the match might fail if the UNIX system does not support multibyte characters in login ID names. In this situation, if you create a database object without explicitly specifying an owner name, the owner name defaults to the UNIX login ID. It will attempt to reference the same database object by qualifying its identifier with an owner name that includes multibyte characters and fail because a string of only single-byte characters cannot match any string containing multibyte characters.

In some East Asian locales, an owner name can include multibyte characters when you create database objects and specify an explicit owner. For example, you can assign an owner name that contains multibyte characters when you specify the owner of an index (within quotation marks) in a CREATE INDEX statement. The following statement declares an index with a multibyte owner name. In this example, the owner name consists of three 2-byte characters:

```
CREATE INDEX 'A1A2B1B2C1C2'.myidx ON mytable (mycol)
```

The preceding example assumes that the client locale supports a multibyte code set and that A^1A^2, B^1B^2, and C^1C^2 are valid characters in this code set.

Path names and file names

Valid path names and file names are operating system-dependent. Multibyte characters in hard-coded path names, for example, limits the portability of your application to operating systems that can support multibyte file names.

Delimited identifiers

A delimited identifier is an identifier that is enclosed in double quotation marks.

When the **DELIMIDENT** environment variable is set, the database server interprets strings of characters in double quotation marks (") as delimited identifiers and strings of characters in single quotation marks (') as data strings. This interpretation of single and double quotation marks is compliant with the ANSI/ISO standard for SQL.

In a nondefault locale, you can specify valid non-ASCII characters of the current code set in most delimited identifiers. You can put non-ASCII characters in a delimited identifier if you can put non-ASCII characters in the undelimited form of the same identifier.

For example, Table 3-1 on page 3-2 indicates that you can specify non-ASCII characters in the declaration of an index name. Thus, you can include non-ASCII characters in an undelimited index name, or in an index name that you have enclosed in double quotation marks to make it a delimited identifier, as in the following SQL statement:

```
CREATE INDEX "A1A2#B1B2" ON mytable (mycol)
```

For a description of delimited identifiers, see the Identifier segment in the *IBM Informix Guide to SQL: Syntax*.

Valid characters in identifiers

In an SQL identifier, a *letter* can be any character in the alpha class that the locale defines. The alpha class lists all characters that are classified as alphabetic.

For more information about character classification, see "The CTYPE category" on page A-3. In the default locale, the alpha class of the code set includes the ASCII characters in the ranges `a` to `z` and `A` to `Z`. SQL identifiers can use these ASCII characters wherever letter is valid in an SQL identifier.

In a nondefault locale, the alpha class of the locale also includes the ASCII characters in the ranges `a` to `z` and `A` to `Z`. It might also include non-ASCII characters, such as letters from non-Roman alphabets (such as Greek or Cyrillic) or ideographic characters. For example, the alpha class of the Japanese UJIS code set (in the Japanese UJIS locale) contains Kanji characters. When IBM Informix products use a nondefault locale, SQL identifiers can use non-ASCII characters wherever letter is valid in the syntax of an SQL identifier. A non-ASCII character is also valid for letter as long as this character is listed in the alpha class of the locale.

The SQL statements in the following example use non-ASCII characters as letters in SQL identifiers:

```
CREATE DATABASE marché;

CREATE TABLE équipement
   (
   code NCHAR(6),
   description NVARCHAR(128,10),
   prix_courant MONEY(6,2)
    );

CREATE VIEW çà_va AS
   SELECT numéro, nom FROM abonnés;
```

In this example, the user creates the following database, table, and view with French-language character names in a French locale (such as **fr_fr.8859-1**):

- The CREATE DATABASE statement declares the identifier **marché**, which includes the 8-bit character **é**, for the database.
- The CREATE TABLE statement declares the identifier **équipement**, which includes the 8-bit character **é**, for the table, and the identifiers **code**, **description**, and **prix_courant** for the columns.
- The CREATE VIEW statement declares the identifier **çà_va**, which includes the 8-bit characters **ç** and **à**, for the view.
- The SELECT clause within the CREATE VIEW statement specifies the identifiers **numéro** and **nom** as columns in the projection list, and the identifier **abonnés** for the table in the FROM clause. Both **numéro** and **abonnés** include the 8-bit character **é**.

All of the identifiers in this example conform to the rules for specifying identifiers within a French locale. For these names to be valid, the database locale must support a code set that includes these French characters in its alpha class.

For the syntax and usage of identifiers in SQL statements, see the Identifier segment in the *IBM Informix Guide to SQL: Syntax*.

Character data types

The locale affects the collation of built-in SQL character data types.

The SQL character data types are:

- Character types that use localized collation: NCHAR and NVARCHAR
- Character types that use code-set order for collation:
  - CHAR
  - LVARCHAR
  - VARCHAR
  - TEXT

The *IBM Informix Guide to SQL: Reference* describes these types. For information about collation, see "Character classes of the code set" on page 1-10.

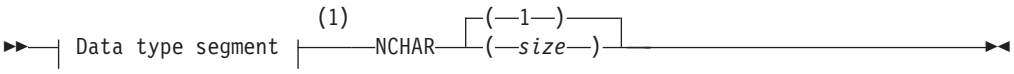
Localized collation of character data

The choice of locale can affect the collating order of NCHAR and NVARCHAR character data types.

The NCHAR data type

The NCHAR data type stores character data in a fixed-length field as a string of single-byte or multibyte letters, numbers, and other characters that are supported by the code set of your database locale.

The syntax of the NCHAR data type is as follows.



Notes:

1 See *IBM Informix Guide to SQL: Syntax*.

Element
Description

size Specifies the number of bytes in the column. The total length of an NCHAR column cannot exceed 32,767 bytes. If you do not specify *size*, the default is NCHAR(1).

Because the length of this column is fixed, when the database server retrieves or sends an NCHAR value, it transfers exactly *size* bytes of data. If the length of a character string is shorter than *size*, the database server extends the string with spaces to make up the *size* bytes. If the string is longer than *size* bytes, the database server truncates the string.

Collate NCHAR data:

NCHAR is a locale-sensitive data type. The only difference between NCHAR and CHAR data types is the collation order, except in databases that were created with the NLSCASE INSENSITIVE property.

The database server sorts data in NCHAR columns in localized order, if the locale defines a localized order. In contrast, the database server collates data in CHAR

columns in code-set order for most operations, even if the database locale (or the SET COLLATION statement of SQL) defines a localized collation.

Because the default locale (U.S. English) has no localized order, the database server sorts NCHAR data in code-set order in the default locale, just as it sorts CHAR data.

Important: In an NLSCASE INSENSITIVE database, strings of all character data types are stored with the same uppercase or lowercase letters as in the records that were loaded or inserted into the database tables. Database server operations on NCHAR and NVARCHAR character strings, however, ignore the case of letters, ordering their data values during collation without respect to or preference for case. Thus, the NCHAR string `"CA"` might precede or follow `"ca"` or `"Ca"` in a collated list, depending on the order in which the data values are retrieved.

Handle NCHAR data:

A client application manipulates NCHAR data by using the **CLIENT_LOCALE** setting of the client system.

The client application performs code-set conversion of NCHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**.

Multibyte characters with NCHAR:

To store multibyte character data in an NCHAR column, your database locale must support a code set that includes the same multibyte characters.

When you store multibyte characters, make sure to calculate the number of bytes that are needed. The *size* parameter of the NCHAR data type refers to the number of bytes of storage that is reserved for the data, rather than to the number of logical characters.

IBM Informix supports the SQL_LOGICAL_CHAR configuration parameter, which can enable logical-character semantics in the declarations of NCHAR and other built-in character data types. For more information, see "Data definition statements" on page 3-36.

Because one multibyte character requires several bytes for storage, the value of *size* bytes does not indicate the number of characters that the column can hold. The total number of multibyte characters that you can store in the column is less than the total number of bytes that you can store in the column. Make sure to declare the *size* value of the NCHAR column in such a way that it can hold enough characters for your purposes.

Treat NCHAR values as numeric values:

If you plan to perform calculations on numbers that are stored in a column, assign a numeric data type (such as INTEGER or FLOAT) to that column.

The description of the CHAR data type in the *IBM Informix Guide to SQL: Reference* provides detailed reasons you do not store certain numeric values in CHAR values. The same reasons apply for certain numeric values as NCHAR values. Treat only numbers that have leading zeros (such as postal codes) as NCHAR data types. Use NCHAR only if you must sort the numeric values in localized order.

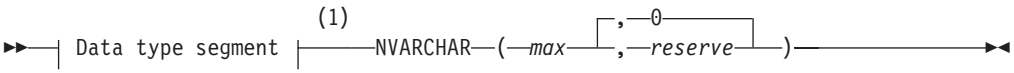
Nonprintable characters with NCHAR:

An NCHAR value can include tabs, spaces, and other white space and nonprintable characters. Nonprintable NCHAR and CHAR values are entered, displayed, and treated similarly.

The NVARCHAR data type

The NVARCHAR data type stores character data in a variable-length field. Data can be a string of single-byte or multibyte letters, digits, and other characters that are supported by the code set of your database locale.

The syntax of the NVARCHAR data type is as follows:



Notes:

1 See *IBM Informix Guide to SQL: Syntax*.

Element
Description

max Specifies the maximum number of bytes that can be stored in the column.

reserve Specifies the minimum number of bytes that can be stored in the column.

You must specify *max* of the NVARCHAR column. The size of this parameter cannot exceed 255 bytes.

When you place an index on an NVARCHAR column, the maximum size is 254 bytes. You can store shorter, but not longer, character strings than the value that you specify.

Specify the *reserve* parameter when you initially intend to insert rows with data values having few or no characters in this column but later expect the data to be updated with longer values. This value can range from 0 to 255 bytes but must be less than the *max* size of the NVARCHAR column. If you do not specify a minimum space value, the default value of *reserve* is 0.

Although use of NVARCHAR economizes on space that is used in a table, it has no effect on the size of an index. In an index that is based on an NVARCHAR column, each index key has a length equal to *max* bytes, the maximum size of the column.

The database server does not strip an NVARCHAR object of any user-entered trailing white space, nor does it pad the NVARCHAR object to the full length of the column. However, if you specify a minimum reserved space (*reserve*), and some of the data values are shorter than that amount, some of the space that is reserved for rows goes unused.

Collate NVARCHAR data:

The NVARCHAR data type is a locale-sensitive data type. The only difference between NVARCHAR and VARCHAR data types is the collation order, except in databases that were created with the NLSCASE INSENSITIVE property.

The database server collates data in NVARCHAR columns in localized order, if the database locale defines a localized order. In contrast, the database server collates data in VARCHAR columns in code-set order for most operations, even if the database locale (or the SET COLLATION statement of SQL) defines a localized collation.

Because the default locale (U.S. English) has no localized order, the database server sorts NVARCHAR data in code-set order in the default locale, just as it sorts VARCHAR data.

Important: In an NLSCASE INSENSITIVE database, strings of all character data types are stored with the same uppercase or lowercase letters as in the records that were loaded or inserted into the database tables. Database server operations on NVARCHAR and NCHAR character strings, however, ignore the case of letters, ordering their data values during collation without respect to or preference for case. Thus, the NVARCHAR string `"VC"` might precede or follow `"vc"` or `"Vc"` in a collated list, depending on the order in which the data values are retrieved.

Handle NVARCHAR data:

Within a client application, always manipulate NVARCHAR data in the **CLIENT_LOCALE** of the client application.

The client application performs code-set conversion of NVARCHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**. (For information about code-set conversion, see "Perform code-set conversion" on page 1-29.)

Multibyte characters with NVARCHAR:

To store multibyte character data in an NVARCHAR column, your database locale must support a code set with these same multibyte characters.

When you store multibyte characters, make sure to calculate the number of bytes that are needed. The *max* parameter of the NVARCHAR data type refers to the maximum number of bytes that the column can store.

Because one multibyte character uses several bytes for storage, the value of *max* bytes does not indicate the number of logical characters that the column can hold. The total number of multibyte characters that you can store in the column is less than the total number of bytes that the column can store. Make sure to declare the *max* value of the NVARCHAR column so that it can hold enough multibyte characters for your purposes.

IBM Informix supports the SQL_LOGICAL_CHAR configuration parameter, which can enable logical-character semantics in the declarations of NVARCHAR and other built-in character data types. For more information, see "Data definition statements" on page 3-36.

Nonprintable characters with NVARCHAR:

An NVARCHAR value can include tabs, spaces, and nonprintable characters. Nonprintable NVARCHAR characters are entered, displayed, and treated in the same way as nonprintable VARCHAR characters.

Tip: The database server interprets the NULL character (ASCII 0) as a C null terminator. In NVARCHAR data, the null terminator acts as a string-terminator character.

Store numeric values in an NVARCHAR column:

The database server does not pad a numeric value in an NVARCHAR column with trailing blanks up to the maximum length of the column.

The number of digits in a numeric NVARCHAR value is the number of characters that you must store that value. For example, the database server stores a value of 1 in the **mytab** table when it executes the following SQL statements:

```
CREATE TABLE mytab (col1 NVARCHAR(10));
INSERT INTO mytab VALUES (1);
```

Performance considerations

The NCHAR data type is like the CHAR data type, and NVARCHAR is like the VARCHAR data type.

These data types differ in two ways:

- The database server collates NCHAR and NVARCHAR column values in localized order.
- The database server collates CHAR and VARCHAR column values in code-set order.

Localized collation depends on the sorting rules that the locale defines, not on the computer representation of the character (the code points). This difference means that the database server might perform complex processing to compare and collate NCHAR and NVARCHAR data. Therefore, access to NCHAR data might be slower with respect to comparison and collation than to access CHAR data. Similarly, access to data in an NVARCHAR column might be slower with respect to comparison and collation than access to the same data in a VARCHAR column.

Assess whether your character data must take advantage of localized order for collation and comparison. If code-set order is adequate, use the CHAR, LVARCHAR, and VARCHAR data types.

Other character data types

The choice of locale can affect the character data types.

The CHAR data type

The CHAR data type stores character data in a fixed-length field. Data can be a string of single-byte or multibyte letters, numbers, and other characters that are supported by the code set of your database locale.

The following list summarizes how the choice of a locale affects the CHAR data type:

- The size of a CHAR column is byte-based, not character-based.

  For example, if you define a CHAR column as CHAR(10), the column has a fixed length of 10 bytes, not 10 characters. If you want to store multibyte characters in a CHAR column, keep in mind that the total number of characters you can store in the column might be less than the total number of bytes you can store in the column. Make sure to define the byte size of the CHAR column so that it can hold enough characters for your purposes.

- You can enter single-byte or multibyte characters in a CHAR column.

  The database locale must support the characters that you want to store in CHAR columns.
- The database server sorts CHAR columns in code-set order, not in localized order.
- Within a client application, always manipulate CHAR data in the **CLIENT_LOCALE** of the client application.

  The client application performs code-set conversion of CHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**.

The VARCHAR data type

The VARCHAR data type stores character strings of up to 255 bytes in a variable-length field. Data can consist of letters, numbers, and symbols. CHARACTER VARYING is handled the same as VARCHAR.

The following list summarizes how the choice of a locale affects the VARCHAR data type:

- The maximum size and minimum reserved space for a VARCHAR column are byte based, not character based.

  For example, if you define a VARCHAR column as VARCHAR(10,6), the column has a maximum length of 10 bytes and a minimum reserved space of 6 bytes. If you want to store multibyte characters in a VARCHAR column, keep in mind that the total number of characters you can store in the column might be less than the total number of bytes you can store in the column. Make sure to define the maximum byte size of the VARCHAR column so that it can hold enough characters for your purposes.
- You can enter single-byte or multibyte characters in a VARCHAR column.

  The database locale must support the characters that you want to store in VARCHAR columns.
- The database server sorts VARCHAR columns in code-set order, not in localized order.
- Within a client application, always manipulate VARCHAR data in the **CLIENT_LOCALE** of the client application.

  The client application performs code-set conversion of VARCHAR data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**.

The LVARCHAR data type

The LVARCHAR data type can store character strings of up to 32,739 bytes in a variable-length field. If you specify no *maximum size* in its declaration, the default upper size limit is 2048 bytes. Data values can include letters, numbers, symbols, white space, and unprintable characters.

LVARCHAR is like the VARCHAR data type in several ways:

- Strings of the LVARCHAR data type are collated in code-set order.
- Client applications perform code-set conversion on LVARCHAR data.
- LVARCHAR supports the built-in SQL length functions. (See "SQL length functions" on page 3-27.)
- LVARCHAR data type declarations can specify a *maximum size*.

Unlike VARCHAR, however, LVARCHAR has no *reserved size* parameter, and data strings in LVARCHAR columns can be longer than the VARCHAR limit of 255 bytes.

The database server also uses LVARCHAR to represent the external format of opaque data types. In I/O operations of the database server, LVARCHAR data values have no upper limit on their size, apart from file size restrictions or limits of your operating system or hardware resources.

The TEXT data type

The TEXT data type stores any text data. TEXT columns typically store memos, manual chapters, business documents, program source files, and other types of textual information.

The following list summarizes how the choice of a locale affects the TEXT data type:

- The database server stores character data in a TEXT column in the code set of the database locale.
- You can enter single-byte or multibyte characters in a TEXT column.

  The database locale supports the characters that you want to store in TEXT columns. However, you can put any type of character in a TEXT column.
- TEXT columns do not have an associated collation order.

  The database server does not build indexes on TEXT columns. Therefore, it does not perform collation tasks on these columns.
- Within a client application, always manipulate TEXT data in the **CLIENT_LOCALE** of the client application.

  The client application performs code-set conversion of TEXT data automatically if **CLIENT_LOCALE** differs from **DB_LOCALE**.

Handle character data

The GLS feature allows you to put non-ASCII characters (including multibyte characters) in the elements of an SQL statement.

You can put non-ASCII characters (including multibyte characters) in the following elements of an SQL statement:

- Quoted strings
- Comments
- Column substrings
- TRIM function arguments
- UPPER, LOWER, and INITCAP function arguments

Specify quoted strings

You use quoted strings in various SQL statements, particularly data manipulation statements such as SELECT and INSERT.

A quoted string is a string of consecutive characters that is delimited by quotation marks. The marks can be single quotation marks or double quotation marks. If the **DELIMIDENT** environment variable is set, however, the database server interprets a string of characters in double quotation marks as a delimited identifier rather than as a string. For more information about delimited identifiers, see “Non-ASCII characters in identifiers” on page 3-1.

When you use a nondefault locale, you can use any characters in the code set of your locale within a quoted string. If the locale supports a code set with non-ASCII

characters, you can use these characters in a quoted string. In the following example, the user inserts column values that include multibyte characters in the table **mytable**:

```
INSERT INTO mytable
   VALUES ('A1A2B1B2abcd', '123X1X2Y1Y2', 'efgh')
```

In this example, the first quoted string includes the multibyte characters A^1A^2 and B^1B^2. The second quoted string includes the multibyte characters X^1X^2 and Y^1Y^2. The third quoted string contains only single-byte characters. This example assumes that the locale supports a multibyte code set with the A^1A^2, B^1B^2, X^1X^2, and Y^1Y^2 characters.

For a description of quoted strings, see the Quoted String segment in the *IBM Informix Guide to SQL: Syntax*.

Specify comments

To use comments after SQL statements, you must use a comment indicator.

Introduce the comment text with one of the following comment indicators:

- The double-hyphen (--) complies with the ANSI SQL standard.
- Braces ({ ... }) are an IBM Informix extension to the ANSI standard.
- C-style slash-and-asterisk (/* . . . */) complies with the SQL-99 standard.

In a nondefault locale, you can use any characters in the code set of your locale within a comment. If the locale supports a code set with non-ASCII characters, you can use these characters in an SQL comment.

In the following example, the user inserts a column value that includes multibyte characters in the table **mytable**:

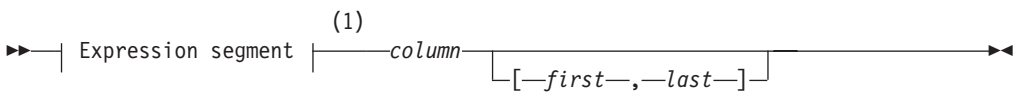
```
EXEC SQL insert into mytable
   values ('A1A2B1B2abcd', '123')   -- A1A2 and B1B2 are multibyte characters.
```

In this example, the SQL comment includes the multibyte characters A^1A^2 and B^1B^2. This example assumes that the locale supports a multibyte code set that includes the A^1A^2 and B^1B^2 characters. For more information about SQL comments and comment indicators, see the *IBM Informix Guide to SQL: Syntax*.

Specify column substrings

In a query (or in any SQL statement containing an embedded SELECT statement), you can use bracket ([ ]) symbols to specify that only a subset of the data in a column of a character data type is to be retrieved. A column expression that includes brackets to signify a subset of the data in the column is known as a *column substring*.

The syntax of a column substring is as follows.



Notes:

1 See *IBM Informix Guide to SQL: Syntax*.

Element
Description

column
Identifier of a column within a database table or view

first. last
Positions of the first and the last byte of the retrieved substring

Column substrings in single-byte code sets

You can use column substrings in single-byte code sets.

Suppose that you want to retrieve the **customer_num** column and the seventh through ninth bytes of the **lname** column from the **customer** table. To perform this query, use a column substring for the **lname** column in your SELECT statement, as follows:

```
SELECT customer_num, lname[7,9] as lname_subset
   FROM customer WHERE lname = 'Albertson'
```

If the **lname** column value is `Albertson`, the query returns these results.

customer_num	lname_subset
114	son

Because the locale supports a single-byte code set, the preceding query seems to return the seventh through ninth characters of the name `Albertson`. Column substrings, however, are byte based, and the query returns the seventh through ninth bytes of the name. Because one byte is equal to one character in single-byte code sets, the distinction between characters and bytes in column substrings is not apparent in these code sets.

Column substrings in multibyte code sets

For multibyte code sets, column substrings return the specified number of bytes, not the number of characters.

If a character column **multi_col** contains a string of three 2-byte characters, this 6-byte string can be represented as follows:

$A^1A^2B^1B^2C^1C^2$

Suppose that a query specified this substring from the **multi_col** column:

```
multi_col[1,2]
```

The query returns the following result:

A^1A^2

The returned substring consists of 2 bytes (one character), not two characters.

To retrieve the first two characters from the **multi_col** column, specify a substring in which *first* is the position of the first byte in the first character and *last* is the position of the last byte in the second character. For the 6-byte string $A^1A^2B^1B^2C^1C^2$, this g expression specifies the substring in your query:

```
multi_col[1,4]
```

The following result is returned:

$A^1A^2B^1B^2$

The substring that the query returns consists of the first 4 bytes of the column value, representing the first two logical characters in the column.

Partial characters in column substrings

A multibyte character might consist of 2, 3, or 4 bytes. A multibyte character that has lost one or more of its bytes so that the original intended meaning of the character is lost is called a *partial character*.

Unless prevented, a column substring might truncate a multibyte character or split it up in such a manner that it no longer retains the original sequence of bytes. A partial character might be generated when you use column subscript operators on columns that contain multibyte characters. Suppose that a user specifies the following column substring for the **multi_col** column where the value of the string in **multi_col** is $A^1A^2B^1B^2C^1C^2$:

```
multi_col[2,5]
```

The user requests the following bytes in the query: $A^2B^1B^2C^1$. If the database server returned this column substring to the user, however, the first and third logical characters in the column would be truncated.

Avoidance in a multibyte code set:

IBM Informix database servers do not allow partial characters to occur. The GLS feature prevents the database server from returning the specified range of bytes literally when this range contains partial characters.

If your database locale supports a multibyte code set and you specify a particular column substring in a query, the database server replaces any truncated multibyte characters with single-byte white space characters.

For example, suppose the **multi_col** column contains the string $A^1A^2A^3A^4B^1B^2B^3B^4$, and you execute the following SELECT statement:

```
SELECT multi_col FROM tablename WHERE multi_col[2,4] = 'A1A2B1B2'
```

The query returns no rows because the database server converts the substring **multi_col[2,4]**, namely the string $A^2A^3A^4$, to three single-byte blank spaces (sss). The WHERE clause specifies this search condition:

```
WHERE 'sss' = 'A1A2A3'
```

Because this condition is never true, the query retrieves no matching rows.

IBM Informix database servers replace partial characters in each individual substring operation, even when they are concatenated.

For example, suppose the **multi_col** column contains $A^1A^2B^1B^2C^1C^2D^1D^2$, and the WHERE clause contains the following condition:

```
multi_col[2,4] | multi_col[6,8]
```

The query does not return any rows because the result of the concatenation ($A^2B^1B^2C^2D^1D^2$) contains two partial characters, A^2 and C^2. The IBM Informix database server converts these partial characters to single-byte blank spaces and creates the following WHERE clause condition:

```
WHERE 'sB1B2sD1D2' = 'A1A2B1B2'
```

This condition is also never true, so the query retrieves no matching rows.

Errors involving partial characters

Partial characters violate the relational model if the substrings strings can be processed or presented to users in any way that can prevent the concatenation of the substrings from reconstructing the original logical string.

This can occur when a multibyte character has a substring that is a valid character by itself. For example, suppose a multibyte code set contains a four-byte character, $A^1A^2A^3A^4$, that represents the digit 1 and a three-byte character, $A^2A^3A^4$, that represents the digit 6. Suppose also that your locale is using this multibyte code set when you execute the following query:

```
SELECT multi_col FROM tablename WHERE multi_col[2,4] = 'A2A3A4'
```

The database server interprets **multi_col[2,4]** as the valid three-byte character (a multibyte 6) instead of a substring of the valid four-byte character ('sss').

Therefore, the WHERE clause contains the following condition:

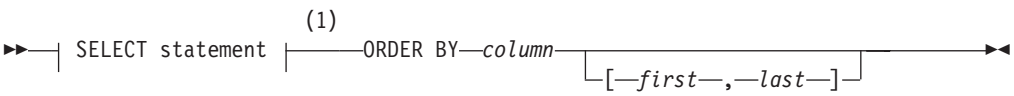
```
WHERE '6' = '6'
```

Partial characters do not occur in single-byte code sets because each character is stored in a single byte. If the database locale supports a single-byte code set, and you specify a column substring in a query, the query returns exactly the requested subset of data; no characters are replaced with white space.

Partial characters in an ORDER BY clause

Partial characters might also create a problem when you specify column substrings in an ORDER BY clause of a SELECT statement.

The syntax for specifying column substrings in the ORDER BY clause is as follows.



Notes:

1 See *IBM Informix Guide to SQL: Syntax*.

Element
Description

column Name of a column in the specified table or view.

first. last
Positions of the first and last byte of the substring.

The query results are sorted by the values contained in this column.

In hierarchical queries, you can optionally specify the ORDER SIBLINGS BY clause, which uses similar syntax to sort the rows returned by the CONNECT BY clause for every level of the data hierarchy.

If the locale supports a multibyte code set whose characters are all of the same length, you can use column substrings in an ORDER BY clause. The more typical scenario, however, is that your multibyte code set contains characters with varying lengths. In this case, you might not find it useful to specify column substrings in the ORDER BY clause.

For example, suppose that you want to retrieve all the rows of the **multi_data** table, and sort the results according to a substring defined as the fourth through sixth characters of the **multi_chars** column, by using this query:

```
SELECT * FROM multi_data ORDER BY multi_chars[7,12]
```

If the locale supports a multibyte code set whose characters are all 2 bytes in length, you know that the fourth character in the column begins in byte position 7, and the sixth character in the column ends in byte position 12. The preceding SELECT statement does not generate partial characters.

If the multibyte code set contains a mixture of single-byte characters, 2-byte characters, and 3-byte characters, however, the substring **multi_chars[7,12]** might create partial characters. In this case, you might get unexpected results when you specify a column substring in the ORDER BY clause.

For information about the collation of different types of character data in the ORDER BY clause, see "The ORDER BY clause" on page 3-19. For the complete syntax and usage of the ORDER BY clause (and of the ORDER SIBLINGS BY clause of hierarchical queries that include the CONNECT BY clause), see the SELECT statement in the *IBM Informix Guide to SQL: Syntax*.

Tip: A partial character might also be generated when a SQL API copies multibyte data from one buffer to another. For more information, see "Generate non-ASCII file names" on page 6-3.

Avoidance in TEXT and BYTE columns:

Partial characters are not a problem when you specify a column substring for a column of the TEXT or BYTE data type.

The database server avoids partial characters in TEXT and BYTE columns in the following way:

- Because the database server interprets a BYTE column as a series of bytes, not characters, the splitting of multibyte characters as a result of the byte range that a column substring specifies is not an issue.

  A substring of a BYTE column returns the exact range of bytes that is specified and does not replace any bytes with white space characters.
- The database server interprets a TEXT value as a character string.

  A substring from a TEXT column returns the exact range of bytes that is specified. Attempts to resolve partial characters in TEXT data are resource-intensive, but the database server does not replace any bytes with white space. For more information, see "The TEXT data type" on page 3-12.

Important: The processing and interpretation of TEXT and BYTE data are the responsibility of the client application, which must handle the possibility of partial characters in these operations.

Specify arguments to the TRIM function

The TRIM function is a built-in SQL function that removes leading or trailing pad characters from character strings of 255 or fewer characters. By default, this pad character is ASCII 32, the blank space.

If your locale supports a code set that defines a different white space character, TRIM does not remove this locale-specific blank space from the front or back of a string. If you specify the LEADING, TRAILING, or BOTH keywords for TRIM, you can specify a different pad character.

You cannot, however, specify a non-ASCII character as a pad character, even if your locale supports a code set that defines the non-ASCII character.

Search functions that are not case-sensitive

The SQL search functions UPPER, LOWER, and INITCAP support GLS. They accept multibyte characters in character-type source strings and operate on them.

The returned data type is the same as the type of the source string:

- UPPER converts every letter in a string to uppercase.
- LOWER converts every letter in a string to lowercase.
- INITCAP changes the first letter of a word or series of words to uppercase.

For complete information about these search functions, see the *IBM Informix Guide to SQL: Syntax*.

Collate character data

Collation is the process of sorting data values in columns that have character data types.

For an explanation of collation order and a discussion of the two methods of sorting character data (code-set order and localized order), see "Character classes of the code set" on page 1-10.

By default, the database server sorts strings according to the collation that the **DB_LOCALE** setting implies, and client applications sort according to the **CLIENT_LOCALE** setting, if this setting is different from the **DB_LOCALE** setting.

The SET COLLATION statement of IBM Informix can specify a localized collation different from the **DB_LOCALE** setting for the current session.

See the *IBM Informix Guide to SQL: Syntax* for the syntax of this statement. Database objects that sort strings, such as indexes or triggers, use the collation that was in effect at the time of their creation when they sort NCHAR or NVARCHAR values, if this setting is different from the **DB_LOCALE** setting.

The collation order of the database server affects SQL statements that perform sorting operations, including CREATE INDEX and SELECT statements.

Collation order in CREATE INDEX

The CREATE INDEX statement creates an index on one or more columns of a table. The ASC and DESC keywords in the CREATE INDEX statement control whether the index keys are stored in ascending or descending order.

When you use a nondefault locale, the following locale-specific considerations apply to the CREATE INDEX statement:

- The index keys are stored in code-set order when you create an index on columns of these data types:
  - CHAR
  - LVARCHAR

– VARCHAR

For example, if the database stores its database locale as the Japanese SJIS locale (**ja_jp.sjis**), index keys for a CHAR column in any table of the database are stored in Japanese SJIS code-set order.

- When you create an index on an NCHAR or NVARCHAR column, the index keys are stored in localized order.

  For example, if the database uses the Japanese SJIS locale, index keys for an NCHAR column in any table of the database are stored in the localized order that the **ja_jp.sjis** locale defines.

If the SET COLLATION statement specifies a database locale with localized collation that is different from the **DB_LOCALE** setting, any indexes (and any check constraints) that you then create in the same session always use that localized collation for sorting NCHAR or NVARCHAR strings.

If you use the default locale (U.S. English), the index keys are stored in the code-set order (in ascending or descending order) of the default code set regardless of the data type of the character column. Because the default locale does not define a localized order, the database server that uses this locale (or any other locale that does not define a localized collating order) sorts strings from columns of the following data types in code-set order:

- CHAR
- LVARCHAR
- NCHAR
- NVARCHAR
- VARCHAR

Collation order in SELECT statements

The SELECT statement performs a query and collation can affect the order of parts of the SELECT statement.

Collation order affects the following parts of the SELECT statement:

- The ORDER BY clause
- The relational, BETWEEN, and IN operators of the WHERE clause
- The MATCHES and LIKE conditions of the WHERE clause

The ORDER BY clause:

The ORDER BY clause sorts retrieved rows by the values that are contained in a column or set of columns.

When this clause sorts character columns, the results of the sort depend on the data type of the column, as follows:

- Columns that are sorted in code-set order:
  – CHAR
  – LVARCHAR
  – VARCHAR
- NCHAR and NVARCHAR columns are sorted in localized order.

Assume that you use a nondefault locale for the client and database locale, and you make a query against the table called **abonnés**. This SELECT statement

specifies three columns of CHAR data type in the select list: **numéro** (employee number), **nom** (family name), and **prénom** (given name).

```
SELECT numéro,nom,prénom
   FROM abonnés
   ORDER BY nom;
```

The statement sorts the query results by the values that are contained in the **nom** column. Because the **nom** column that is specified in the ORDER BY clause is a CHAR column, the database server sorts the query results in the code-set order.

As this table shows, names that begin with uppercase letters come before names beginning with lowercase letters, and names that begin with an accented letter (`Ålesund`, `Étaix`, `Ötker`, and `Øverst`) are at the end of the list.

Table 3-2. Data set for code-set order of the abonnés table

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13606	Dupré	Michéle Françoise
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13609	Tiramisù	Paolo Alfredo
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13601	Ålesund	Sverre
13608	Étaix	Émile
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

Results of the query are different, however, if the **numéro**, **nom**, and **prénom** columns of the **abonnés** table are defined as NCHAR rather than CHAR.

Suppose the nondefault locale defines a localized order that collates the data as the following table shows. This localized order defines equivalence classes for uppercase and lowercase letters and for unaccented and accented versions of the same letter.

Table 3-3. Data set for localized order of the abonnés table

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13601	Ålesund	Sverre
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13606	Dupré	Michéle Françoise

Table 3-3. Data set for localized order of the abonnés table (continued)

numéro	nom	prénom
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders
13609	Tiramisù	Paolo Alfredo

The same SELECT statement now returns the query results in localized order because the **nom** column that the ORDER BY clause specifies is an NCHAR column.

The SELECT statement supports use of a column substring in an ORDER BY clause. However, you need to ensure that this use for column substrings works with the code set that your locale supports. For more information, see "Partial characters in column substrings" on page 3-15.

Logical predicates in a WHERE clause:

The WHERE clause specifies search criteria and join conditions on the data that you want to select.

Collation rules affect the WHERE clause when the expressions in the condition are column expressions with character data types and the search condition is one of the following logical predicates:

- Relational-operator condition
- BETWEEN condition
- IN condition
- EXISTS and ANY conditions

Relational-operator conditions:

The example SELECT statement assumes a nondefault locale and uses relational-operator conditions.

It uses the less than (<) relational operator to specify that the only rows are to be retrieved from the **abonnés** table are those in which the value of the **nom** column is less than `Hammer`.

```
SELECT numéro,nom,prénom
   FROM abonnés
   WHERE nom < 'Hammer';
```

If **nom** is a CHAR column, the database server uses code-set order of the default code set to retrieve the rows that the WHERE clause specifies. The output shows that this SELECT statement retrieves only two rows.

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13606	Dupré	Michéle Françoise

These two rows are those rows less than `Hammer` in the code-set-ordered data set shown in Table 3-2 on page 3-20.

However, if **nom** is an NCHAR column, the database server uses localized order to sort the rows that the WHERE clause specifies. The following example of output shows that this SELECT statement retrieves six rows.

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13601	Ålesund	Sverre
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13606	Dupré	Michéle Françoise
13608	Étaix	Émile

These six rows are those rows less than `Hammer` in the localized-order data set shown in Table 3-3 on page 3-20.

BETWEEN conditions:

The example SELECT statement assumes a nondefault locale and uses BETWEEN conditions.

The following SELECT statement uses a BETWEEN condition to retrieve only those rows in which the values of the **nom** column are in the inclusive range of the values of the two expressions that follow the BETWEEN keyword:

```
SELECT numéro,nom,prénom
   FROM abonnés
   WHERE nom BETWEEN 'A' AND 'Z';
```

The query result depends on whether **nom** is a CHAR or NCHAR column. If **nom** is a CHAR column, the database server uses the code-set order of the default code set to retrieve the rows that the WHERE clause specifies. The following example of output shows the query results.

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13606	Dupré	Michéle Françoise
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse

numéro	nom	prénom
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13609	Tiramisù	Paolo Alfredo

Because the database server uses the code-set order for the **nom** values, as Table 3-2 on page 3-20 shows, these query results do not include the following rows:

- Rows in which the value of **nom** begins with a lowercase letter: `da Sousa` and `di Girolamo`
- Rows with an accented letter: `Ålesund`, `Étaix`, `Ötker`, and `Øverst`

However, if **nom** is an NCHAR column, the database server uses localized order to sort the rows. The following output shows the query results.

numro	nom	prnom
13612	Azevedo	Edouardo Freire
13601	Ålesund	Sverre
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13606	Dupré	Michéle Françoise
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders
13609	Tiramisù	Paolo Alfredo

Because the database server uses localized order for the **nom** values, these query results include rows in which the value of **nom** begins with a lowercase letter or accented letter.

IN conditions:

An IN condition is satisfied when the expression to the left of the IN keyword is included in the parenthetical list of values to the right of the keyword.

This SELECT statement assumes a nondefault locale and uses an IN condition to retrieve only those rows in which the value of the **nom** column is any of the following: `Azevedo`, `Llanero`, or `Oatfield`.

```
SELECT numéro,nom,prénom
   FROM abonnés
   WHERE nom IN ('Azevedo', 'Llanero', 'Oatfield');
```

The query result depends on whether **nom** is a CHAR or NCHAR column. If **nom** is a CHAR column, the database server uses code-set order, as Table 3-2 on page 3-20 shows. The database server retrieves rows in which the value of **nom** is `Azevedo`, but not rows in which the value of **nom** is `azevedo` or `Åzevedo` because the characters `A`, `a`, and `Å` are not equivalent in the code-set order. The query also returns rows with the **nom** values of `Llanero` and `Oatfield`.

However, if **nom** is an NCHAR column, the database server uses localized order, as Table 3-3 on page 3-20 shows, to sort the rows. If the locale defines `A`, `a`, and `Å` as equivalent characters in the localized order, the query returns rows in which the value of **nom** is `Azevedo`, `azevedo`, or `Åzevedo`. The same selection rule applies to the other names in the parenthetical list that follows the IN keyword.

Comparisons with MATCHES and LIKE conditions

Collation rules also affect the WHERE clause when the expressions in the condition are column expressions with character data types and the search condition is either the MATCHES or LIKE condition.

MATCHES condition:

A MATCHES condition tests for matching character strings.

The condition is true, or satisfied, when the value of the column to the left of the MATCHES keyword matches the pattern that a quoted string specifies to the right of the MATCHES keyword. You can use wildcard characters in the string. For example, you can use brackets to specify a range of characters. For more information about MATCHES, see the *IBM Informix Guide to SQL: Syntax*.

When a MATCHES expression does not list a range of characters in the string, it specifies a *literal match*. For literal matches, the data type of the column determines whether collation considerations come into play, as follows:

- For CHAR and VARCHAR columns, no collation considerations come into play.
- For NCHAR and NVARCHAR columns, collation considerations might come into play, because these data types use localized order and the locale might define equivalence classes of collation.

  For example, the localized order might specify that `a` and `A` are an equivalent class. That is, they have the same rank in the collation order. For more information about localized order, see "Localized order" on page 1-12.

The examples in the following table illustrate the different results that CHAR and NCHAR columns produce when a user specifies the MATCHES keyword without a range in a SELECT statement. These examples assume use of a nondefault locale that defines `A` and `a` in an equivalence class. It also assumes that **col1** is a CHAR column and **col2** is an NCHAR column in table **mytable**.

Query	Data type	Query results
SELECT * FROM mytable WHERE col1 MATCHES 'art'	CHAR	All rows in which column **col1** contains the value 'art' with a lowercase *a*
SELECT * FROM mytable WHERE col2 MATCHES 'art'	NCHAR	All rows in which column **col2** contains the value 'art' or 'Art'

When you use the MATCHES keyword to specify a range, collation considerations come into play for all columns with character data types. When the column to the left of the MATCHES keyword is an NCHAR, NVARCHAR, CHAR, VARCHAR, or LVARCHAR data type, and the string operand of the MATCHES keyword includes brackets ([ ]) to specify a range, sorting follows a localized order, if the locale defines one.

Important: When the database server determines the characters that fall within a range with the MATCHES operator, it uses the localized order, if **DB_LOCALE** or SET COLLATION has specified one, even for CHAR, LVARCHAR, and VARCHAR columns. This behavior is an exception to the rule that the database server uses code-set order for all operations on CHAR, LVARCHAR and VARCHAR columns, and localized order (if one is defined) for sorting operations on NCHAR and NVARCHAR columns.

Some simple examples show how the database server treats NCHAR, NVARCHAR, LVARCHAR, CHAR, and VARCHAR columns when you use the MATCHES keyword with a range in a SELECT statement. Suppose that you want to retrieve from the **abonnés** table the employee number, given name, and family name for all employees whose family name **nom** begins in the range of characters `E` through `P`. Also assume that the **nom** column is an NCHAR column. The following SELECT statement uses a MATCHES condition in the WHERE clause to pose this query:

```
SELECT numéro,nom,prénom
   FROM abonnés
   WHERE nom MATCHES '[E-P]*'
   ORDER BY nom;
```

The rows for `Étaix`, `Ötker`, and `Øverst` appear in the query result because, in the localized order, as Table 3-3 on page 3-20 shows, the accented first letter of each name falls within the `E` through `P` MATCHES range for the **nom** column.

numéro	nom	prénom
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

If **nom** is a CHAR column, the query result is the same as when **nom** was an NCHAR column. The database server always uses localized order to determine what characters fall within a range, regardless of whether the column is CHAR or NCHAR.

LIKE condition:

A LIKE condition tests for matching character strings.

As with the MATCHES condition, the LIKE condition is true, or satisfied, when the value of the column to the left of the LIKE keyword matches the pattern that the quoted string specifies to the right of the LIKE keyword. You can use only certain symbols as wildcards in the quoted string. For more information about LIKE, see the *IBM Informix Guide to SQL: Syntax*.

The LIKE condition can specify only a *literal match*. For literal matches, the data type of the column determines whether collation considerations come into play, as follows:

- For CHAR and VARCHAR columns, no collation considerations come into play.
- For NCHAR and NVARCHAR columns, collation considerations might come into play because these data types use localized order, and the locale might define equivalence classes of collation. For example, the localized order might specify that `a` and `A` are an equivalent class.

The LIKE keyword does not support ranges of characters. That is, you cannot use bracketed characters to specify a range in LIKE conditions.

Wildcard characters in LIKE and MATCHES conditions:

IBM Informix products support ASCII characters as wildcard characters in the MATCHES and LIKE conditions.

IBM Informix products support the following ASCII characters as wildcard characters:

Condition	Wildcard characters
LIKE	_ %
MATCHES	* ? [ ] ^ -

For CHAR and VARCHAR data, the database server performs byte-by-byte comparison for pattern matching in the LIKE and MATCHES conditions. For NCHAR and NVARCHAR data, the database server performs pattern matching in the LIKE and MATCHES conditions based on logical characters, not bytes. Therefore, the underscore (_) wildcard of the LIKE clause and the ? (question mark) wildcard of the MATCHES clause match any one single-byte or multibyte character, as the following table shows.

Condition	Quoted string	Column value	Result
LIKE	'ab_d'	'abcd'	True
LIKE	'ab_d'	'abA1A2d'	True
MATCHES	'ab?d'	'abcd'	True
MATCHES	'ab?d'	'abA1A2d'	True

The database server treats any multibyte character as a literal character. To tell the database server to interpret a wildcard character as its literal meaning, you must precede the character with an escape character. You must use single-byte characters as escape characters; the database server does not recognize use of multibyte characters for this purpose. The default escape character is the backslash (\) symbol.

The following MATCHES condition returns a TRUE result for the column value that is shown.

Condition	Quoted string	Column value	Result
MATCHES	'ab\?d'	'ab?d'	True

SQL length functions

You can use SQL length functions in the SELECT statement and other data manipulation statements. Length functions return the length of a column, string, or variable in bytes or characters.

For the syntax of these functions, see the Expression segment in the *IBM Informix Guide to SQL: Syntax*.

The choice of locale affects the three SQL length functions.

The LENGTH function

The LENGTH function returns the number of bytes of data in character data.

However, the behavior of the LENGTH function varies with the type of argument that the user specifies. The argument can be a quoted string, a character-type column other than the TEXT data type, a TEXT column, a host variable, or an SPL routine variable.

The following table shows how the LENGTH function operates on each of these argument types. The **Example** column in this table uses the symbol s to represent a single-byte trailing white space character.

This table assumes that all arguments consist of single-byte characters.

LENGTH argument	Behavior	Example
Quoted string	Returns number of bytes in string, minus any trailing white space (as defined in the locale).	If the string is 'Ludwig', the result is 6. If the string is 'Ludwigssss', the result is still 6.
CHAR, VARCHAR, LVARCHAR, NCHAR, or NVARCHAR column	Returns number of bytes in a column, minus any trailing white- space characters, regardless of defined length of the column.	If the **fname** column of the **customer** table is a CHAR(15) column, and this column contains the string 'Ludwig', the result is 6. If the **fname** column contains the string 'Ludwigssss', the result is still 6.
TEXT column	Returns number of bytes in a column, including trailing white space characters.	If the **cat_descr** column in the **catalog** table is a TEXT column, and this column contains the string 'Ludwig', the result is 6. If the **cat_descr** column contains the string 'Ludwigssss', the result is 10.
Host or procedure variable	Returns number of bytes that the variable contains, minus any trailing white pace, regardless of defined length of the variable.	If the procedure variable **f_name** is defined as CHAR(15), and this variable contains the string 'Ludwig', the result is 6. If the **f_name** variable contains the string 'Ludwigssss', the result is still 6.

When you use the default locale or any locale with a single-byte code set, the LENGTH function seems to return the number of characters in the column. In the following example, the **stores_demo** database, which contains the **customer** table, uses the default code set for the U.S. English locale. Suppose a user enters a SELECT statement with the LENGTH function to display the family name, length of the family name, and customer number for rows where the customer number is less than 106.

```
SELECT lname AS cust_name,
   length (fname) AS length, customer_num AS cust_num
   FROM customer WHERE customer_num < 106
```

The following example of output shows the result of the query. For each row that is retrieved, the **length** column seems to show the number of characters in the **lname** (**cust_name**) column. However, the **length** column actually displays the number of bytes in the **lname** column.

In the default code set, one byte stores one character. For more information about the default code set, see "The default locale" on page 1-21.

cust_name	length	cust_num
Ludwig	6	101
Carole	6	102
Philip	6	103
Anthony	7	104
Raymond	7	105

When you use the LENGTH function in a locale that supports a multibyte code set, such as the Japanese SJIS code set, the distinction between characters and bytes is meaningful. LENGTH returns the number of bytes in its argument. This result might be different from the number of characters.

The next example assumes that the database that contains the **customer_multi** table has locale with a multibyte code set. Suppose that the user enters a SELECT statement with the LENGTH function to display **lname**, its length, and **customer_num** for the customer whose number is 199.

```
SELECT lname AS cust_name,
   length (fname) AS length, customer_num AS cust_num
   FROM customer_multi WHERE customer_num = 199
```

Suppose that **lname** for customer 199 consists of four characters:

aA^{1}A^{2}bB^{1}B^{2}

In this representation, the first character (the symbol a) is a single-byte character. The second character (the symbol A1A2) is a 2-byte character. The third character (the symbol b) is a single-byte character. The fourth character (the symbol B1B2) is a 2-byte character.

The following example of output shows the result of the query. Although the customer given name consists of four characters, the length column shows that the total number of bytes in this name is 6.

cust_name	length	cust_num
aA^{1}A^{2}bB^{1}B^{2}	6	199

The OCTET_LENGTH function

The OCTET_LENGTH function returns the number of bytes and generally includes trailing white space characters in the byte count.

This SQL length function uses the definition of white space that the locale defines. OCTET_LENGTH returns the number of bytes in a character column, quoted string, host variable, or SPL variable. The actual behavior of OCTET_LENGTH varies with the type of argument that the user specifies.

The following table shows how the OCTET_LENGTH function operates on each of the argument types. The Example column in this table uses the symbol s to represent a single-byte trailing white space character. For simplicity, the Example column also assumes that the example strings consist of single-byte characters.

OCTET_LENGTH argument	**Behavior**	**Example**
Quoted string	Returns number of bytes in string, including any trailing white- space characters.	If the string is 'Ludwig', the result is 6. If the string is 'Ludwigssss', the result is 10.
CHAR or NCHAR column	Returns number of bytes in string, including trailing white space characters. This value is the defined length, in bytes, of the column.	If the **fname** column of the **customer** table is a CHAR(15) column, and this column contains the string 'Ludwig', the result is 15. If the **fname** column contains the string 'Ludwigsss', the result is still 15.
VARCHAR or NVARCHAR column	Returns number of bytes in string, including trailing white space. Value is the actual length, in bytes, of the character string, not the declared maximum column size.	If the **cat_advert** column of the catalog table is a VARCHAR(255, 65) column, and this column contains the string "Ludwig", the result is 6. If the column contains the string 'Ludwigssss', the result is 10.
TEXT column	Returns number of bytes in column, including trailing white- space characters.	If the **cat_descr** column in the **catalog** table is a TEXT column, and this column contains the string 'Ludwig', the result is 6. If the **cat_descr** column contains the string 'Ludwigssss', the result is 10.
Host or procedure variable	Returns number of bytes that the variable contains, including any trailing white space, regardless of defined length of variable.	If the procedure variable **f_name** is defined as CHAR(15), and this variable contains the string 'Ludwig', the result is 6. If the **f_name** variable contains the string 'Ludwigssss', the result is 10.

The difference between the LENGTH and OCTET_LENGTH functions is that OCTET_LENGTH generally includes trailing white space in the byte count, whereas LENGTH generally excludes trailing white space from the byte count.

The advantage of the OCTET_LENGTH function over the LENGTH function is that the OCTET_LENGTH function provides the actual column size whereas the LENGTH function trims the column values and returns the length of the trimmed string. This advantage of the OCTET_LENGTH function applies both to single-byte code sets such as ISO8859-1 and multibyte code sets such as the Japanese SJIS code set.

The following table shows some results that the OCTET_LENGTH function might generate.

OCTET_LENGTH input string	Description	Result
'abc '	A quoted string with four single-byte characters (the characters abc and one trailing space)	4
'A[1]A[2]B[1]B[2]'	A quoted string with two multibyte characters	4
'aA[1]A[2]bB[1]B[2]'	A quoted string with two single-byte and two multibyte characters	6

The CHAR_LENGTH function

The CHAR_LENGTH function (also known as the CHARACTER_LENGTH function) returns the number of characters in a quoted string, column with a character data type, host variable, or procedure variable. However, the actual behavior of this function varies with the type of argument that the user specifies.

The following table shows how the CHAR_LENGTH function operates on each of the argument types. The **Example** column in this table uses the symbol s to represent a single-byte trailing white space. For simplicity, the **Example** column assumes that the strings consist of single-byte characters.

CHAR_LENGTH argument	Behavior	Example
Quoted string	Returns number of characters in string, including any trailing white- space (as defined in the locale).	If the string is 'Ludwig', the result is 6. If the string is 'Ludwigssss', the result is 10.
CHAR or NCHAR column	Returns number of characters in string, including trailing white space characters. This value is the defined length, in bytes, of the column.	If the **fname** column of the **customer** table is a CHAR(15) column, and this column contains the string 'Ludwig', the result is 15. If the **fname** column contains the string 'Ludwigssss', the result is 15.
VARCHAR or NVARCHAR column	Returns number of characters in string, including white space characters. Value is the actual length, in bytes, of the string, not the declared maximum column size.	If the **cat_advert** column of the catalog table is a VARCHAR(255, 65), and this column contains the string "Ludwig", the result is 6. If the column contains the string 'Ludwigssss', the result is 10.
TEXT column	Returns number of characters in column, including trailing white space characters.	If the **cat_descr** column in the **catalog** table is a TEXT column, and this column contains the string 'Ludwig', the result is 6. If the **cat_descr** column contains the string 'Ludwigssss', the result is 10.
Host or procedure variable	Returns number of characters that the variable contains, including any trailing white space, regardless of declared length of the variable.	If the procedure variable **f_name** is defined as CHAR(15), and this variable contains the string 'Ludwig', the result is 6. If the **f_name** variable contains the string 'Ludwigssss', the result is 10.

The CHAR_LENGTH function is especially useful with multibyte code sets. If a quoted string of characters contains any multibyte characters, the number of characters in the string differs from the number of bytes in the string. You can use the CHAR_LENGTH function to determine the number of characters in the quoted string.

However, the CHAR_LENGTH function can also be useful in single-byte code sets. In these code sets, the number of bytes in a column is equal to the number of characters in the column. If you use the LENGTH function to determine the number of bytes in a column (which is equal to the number of characters in this case), LENGTH trims the column values and returns the length of the trimmed string. In contrast, CHAR_LENGTH does not trim the column values but returns the declared size of the column.

The following table shows some results that the CHAR_LENGTH function might generate for quoted strings.

CHAR_LENGTH input string	Description	Result
'abc '	A quoted string with 4 single-byte characters (the characters abc and 1 trailing space)	4
'A1A2B1B2'	A quoted string with 2 multibyte characters	2
'aA1A2B1B2'	A quoted string with 2 single-byte and 2 multibyte characters	4

Locale-sensitive data types

These topics explain how a locale affects the way that a database server handles the MONEY data type, extended data types, and smart large objects (CLOB and BLOB data types).

For the syntax of these data types, see the *IBM Informix Guide to SQL: Syntax*. For descriptions of these data types, see the *IBM Informix Guide to SQL: Reference*.

Handle the MONEY data type

The MONEY data type stores currency amounts. This data type stores fixed-point decimal numbers up to a maximum of 32 significant digits. You can specify MONEY columns in data definition statements such as CREATE TABLE and ALTER TABLE.

The choice of locale affects monetary data in the following ways:

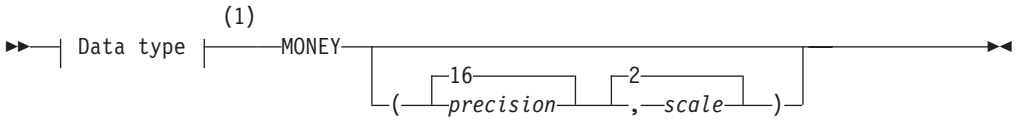
- The default value of *scale* in the declaration of MONEY columns
- The currency notation that the client application uses

The locale defines the default scale and currency notation in the MONETARY category of the locale file. For information about the MONETARY category of the locale file, see "The MONETARY category" on page A-5.

Specify values for the scale parameter

You can define a MONEY column with a syntax.

Define a MONEY column with the following syntax.



Notes:

1 See *IBM Informix Guide to SQL: Syntax*.

Element
Description

precision
Total number of significant digits in a decimal or money data type

You must specify an integer 1 - 32, inclusive. The default *precision* is 16.

scale Number of digits to the right of the decimal point.

The *scale* must be an integer between 1 and *precision*. If you omit the *scale*, the database server provides a default *scale* that the database locale defines. For the default locale (U.S. English), the default is 2, as the diagram indicates.

Internally, the database server stores MONEY values as DECIMAL values. The *precision* parameter defines the total number of significant digits, and the *scale* parameter defines the total number of digits to the right of the decimal separator. For example, if you define a column as MONEY(8,3), the column can contain a maximum of eight digits, and three of these digits are to the right of the decimal separator. An example of a data value in the column might be `12345.678`.

If you omit the *scale* parameter from the declaration of a MONEY column, the database server provides a scale that the locale defines. For the default locale (U.S. English), the database server uses a default scale of `2`. It stores the data type MONEY(*precision*) in the same internal format as the data type DECIMAL(*precision*,2). For example, if you define a column as MONEY(10), the database server creates a column with the same format as the data type DECIMAL(10,2). A data value in the column might be `12345678.90`.

For nondefault locales, if you omit the *scale* when you declare a MONEY column, the database server declares a column with the same internal format as DECIMAL data types with a locale-specific default scale. For example, if you define a column as MONEY(10), and the locale defines the default scale as `4`, the database server stores the data type of the column in the same format as DECIMAL(10,4). A data value in the column might be `123456.7890`.

The GLS code sets for most European languages can support the euro symbol in monetary values. For the complete syntax of the MONEY data type, see the *IBM Informix Guide to SQL: Syntax*. For a complete description of the MONEY data type, see the *IBM Informix Guide to SQL: Reference*.

Format of currency notation

Client applications format values in MONEY columns with the currency notation that the locale defines.

This notation specifies the currency symbol, thousands separator, and decimal separator. For more information about currency notation, see “Numeric and monetary formats” on page 1-15.

In the default locale, the default currency symbol is a dollar sign ($), the default thousands separator is a comma (,), and the default decimal separator is a period (.) symbol. For nondefault locales, the locale defines the appropriate culture-specific currency notation for monetary values. You can also use the **DBMONEY** environment variable to customize the currency symbol and decimal separator for monetary values. For more information, see "Customize monetary values" on page 1-34.

Handle extended data types

The extensible data type system of IBM Informix allows users to define new data types and the behavior of these new data types to the database server.

This section explains how these types are handled in GLS processing. See also *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Opaque data types

An opaque data type is fully encapsulated to client applications; that is, its internal structure is not known to the database server.

Therefore, the database server cannot automatically perform locale-specific tasks such as code-set conversion for opaque types. All GLS processing (code-set conversion, localized collation order, end-user formats, and so on) must be performed in the opaque-type support functions.

When you create an opaque data type, you can write the support functions as C UDRs that can handle any locale-sensitive data. For more information, see "Locale-sensitive data in an opaque data type" on page 4-16.

Complex data types

IBM Informix also supports collection data types (SET, MULTISET, and LIST) and row data types(named ROW types and unnamed ROW types).

Any of these data types can have members with character, time, or numeric data types. The database server can still handle the GLS processing for these data types when they are part of a complex data type.

Distinct data types

A distinct data type has the same internal storage representation as its source type but has a different name. Its source type can be an opaque or built-in type, a named ROW type, or another distinct data type. IBM Informix handles GLS considerations for a distinct type as it would for the source type.

Handle smart large objects

A smart large object can store text or images. Smart large objects are stored and retrieved in pieces and have database properties such as recovery and transaction rollback.

IBM Informix supports two smart-large-object types:

- The BLOB data type stores any type of binary data, including images and video clips.
- The CLOB data type stores text such as PostScript or HTML files.

You can seek smart large objects in bytes but not in characters. Therefore, you need to manage the byte offset of multibyte characters when you search for information in smart large objects.

To access smart large objects through a client application, you must use an API, such as Informix ESQL/C or DataBlade API. Because GLS does not support direct access to smart-large-object data through SQL, GLS does not automatically handle the data (no automatic code-set conversion, localized collation order, end-user formats, and so on). All support must be done within an API.

When you copy CLOB data from a file, IBM Informix performs any necessary character-set conversions. If the client (when it copies from client files) or server locale (when it copies from server files) differs from the database locale, IBM Informix invokes the routines to convert to the database locale.

Data manipulation statements

The choice of a locale can affect certain SQL data manipulation statements.

These SQL data manipulation statements can be affected:

- DELETE
- INSERT
- LOAD
- MERGE
- UNLOAD
- UPDATE

Sections describe the GLS aspects of these SQL statements. For a complete description of these statements, see the *IBM Informix Guide to SQL: Syntax*.

Specify conditions in the WHERE clause

You can specify conditions in the WHERE clause for several statements to specify rows on which to operate.

These statements can include a WHERE clause to specify rows on which to operate:

- For the DELETE statement, the WHERE clause specifies rows to delete.
- For the INSERT or MERGE statement with an embedded SELECT, the WHERE clause specifies which rows to insert from another table.
- For the UPDATE or MERGE statement, the WHERE clause specifies which rows to update. In addition, the SET clause of UPDATE or MERGE can include an embedded SELECT statement whose WHERE clause identifies a row whose values are to be assigned to another row.
- For the UNLOAD feature of DB-Access, the WHERE clause of the embedded SELECT specifies which rows to unload.

The choice of a locale affects these uses of a WHERE clause in the same way that it affects the WHERE clause of a SELECT. For more information, see "Logical predicates in a WHERE clause" on page 3-21 and "Comparisons with MATCHES and LIKE conditions" on page 3-24.

Specify era-based dates

You can specify era-based dates in several SQL statements.

These SQL statements might specify DATE and DATETIME column values:

- The WHERE clause of the DELETE statement

- The VALUES clause of the INSERT or MERGE statement
- The SET clause of the UPDATE or MERGE statement

When you specify a DATE column value in one of the preceding SQL statements, the database server uses the **GL_DATE** (or **DBDATE**) environment variable to interpret the date expression, as follows:

- If you have set **GL_DATE** (or **DBDATE**) to an era-based (Asian) date format, you can use era-based date formats for date expressions.
- If you have not set the **GL_DATE** (or **DBDATE**) environment variable to an era-based date format, you can use era-based date formats for date expressions only if the server-processing locale supports era-based dates. For more information about the server-processing locale, see "Determine the server-processing locale" on page 1-26.
- If your locale does not support era-based dates, you cannot use era-based date formats for date expressions. If you attempt to specify an era-based date format in this case, the SQL statement fails.

When you specify a DATETIME column value, the database server uses the **GL_DATETIME** (or **DBTIME**) environment variable instead of the **GL_DATE** (or **DBDATE**) environment variable to interpret the expression.

For more information, see "Era-based date and time formats" on page 1-33.

Load and unload data

The LOAD and UNLOAD features of DB-Access enable you to transfer data to and from your database with operating-system text files.

The following topics describe the GLS aspects of the LOAD and UNLOAD statements. For a complete description of the use and syntax of these DB-Access features, see the *IBM Informix Guide to SQL: Syntax*.

Load data into a database

The LOAD statement inserts data from an operating-system file into an existing table or view. This operating-system file is called a `LOAD FROM` file.

The data in this file can contain any character that the client code set defines. If the client locale supports a multibyte code set, the data can contain multibyte characters. If the database locale supports a code set that is different from but convertible to the client code set, the client performs code-set conversion on the data before sending the data to the database server. For more information, see "Perform code-set conversion" on page 1-29.

The locale also defines the formats for date, time, numeric, and monetary data. You can apply any format that the client locale supports to column values in the LOAD FROM file. For example, a French locale might define monetary values that have a blank space as the thousands separator and a comma as the decimal separator. When you use this locale, the following literal value for a MONEY column is valid in a LOAD FROM file:

```
3 411,99
```

You can specify alternative formats for date and monetary data. If you set appropriate environment variables, the `LOAD FROM` files can use the alternative end-user formats for DATE, DATETIME, and MONEY column values. For more

information, see “Customize date and time end-user formats” on page 1-32 and “Customize monetary values” on page 1-34.

Unload data from a database

The UNLOAD statement writes the rows that a SELECT statement retrieves to an operating-system file. This operating-system file is called an UNLOAD TO file.

The data values in this file contains characters that the client code set defines. If the client locale supports a multibyte code set, the data can include multibyte characters from the code set.

If the database locale supports a code set that is different from but convertible to the client code set, the client performs code-set conversion on the data before it writes the data to the UNLOAD TO file. (For more information, see “Perform code-set conversion” on page 1-29.)

The client locale and certain environment variables determine the output format of certain data types in the UNLOAD TO file. These data types include DATE values, MONEY values, values of numeric data types, and DATETIME values. For further information, see “End-user formats” on page 1-13 and “Customize end-user formats” on page 1-32.

Important: You can use an UNLOAD TO file, which the UNLOAD statement generates, as the input file (the LOAD FROM file) to a LOAD statement that loads another table or database. When you use an UNLOAD TO file in this manner, make sure that all environment variables and the client locale have the same values when you perform the LOAD as they did when you performed the UNLOAD.

Data definition statements

IBM Informix supports a configuration parameter, SQL_LOGICAL_CHAR, whose setting can simplify the use of certain Data Definition Language (DDL) statements of SQL when you declare character data types in locales that support multibyte code sets.

If the IBM Informix instance has SQL_LOGICAL_CHAR set to enable logical character semantics in declarations of character data types, the maximum number of bytes that are required to store a single character of the code set of the locale can affect these SQL data definition statements:

- ALTER TABLE
- CREATE TABLE

The SQL_LOGICAL_CHAR setting can also affect the DEFINE statement of SPL when it declares character variables.

The SQL_LOGICAL_CHAR feature addresses a potential problem for data management applications that are developed in a single-byte locale, such as the default locale, but that are later deployed in a multibyte locale. By default, numeric size specifications in declarations of character data types are interpreted in units of bytes. A character column that can store strings of up to 10 bytes, for example, can store no more than two logical characters that each requires four bytes of storage. A table schema that was designed for a single-byte locale might lead to data truncation in operations on character strings in multibyte characters.

The setting of the SQL_LOGICAL_CHAR configuration parameter, however, can change the behavior of the SQL parser, so that size specifications in character data type declarations are interpreted in units of logical characters, rather than as bytes. The maximum declared size is multiplied by a numeric factor, as specified by the setting of this parameter.

The following table shows the valid settings and their effects:

Value	Effect
OFF or 1	No expansion of declared sizes
2	Use 2 as the expansion factor for declared sizes.
3	Use 3 as the expansion factor for declared sizes.
4	Use 4 as the expansion factor for declared sizes.
ON	Use *M* as the expansion factor, where *M* is the maximum storage length in bytes that any logical character requires in the code set of the current database. Depending on the code set associated with the **DB_LOCALE** setting, *M* has a positive integer range from 1 (in single-byte locales) up to 4.

When the SQL_LOGICAL_CHAR configuration parameter is set to a value greater than 1, it instructs the SQL parser to interpret explicit and implicit size declarations as logical characters, rather than as bytes, in declarations of SPL variables and in CREATE TABLE and ALTER TABLE statements that define columns of the following data types:

- CHAR and CHARACTER
- CHARACTER VARYING and VARCHAR
- LVARCHAR
- NCHAR
- NVARCHAR
- DISTINCT types whose base types are built in character data types.
- DISTINCT types whose base types are the previously listed data types.
- ROW data type fields of any of the previously listed data types.
- Elements of the previously listed data types within LIST, MULTISET, and SET collection objects.

The SQL_LOGICAL_CHAR setting has no effect, however, on TEXT or CLOB objects, nor on user-defined data types (UDTs) that store character strings.

Enabling logical character semantics for the database locale guarantees that sufficient storage is available for the data type to store the specified number of logical characters. The resulting size in bytes of a character column in a database table or of an SPL character variable is the product of the declared size of the data type multiplied by the SQL_LOGICAL_CHAR value, if this size is 2, 3, or 4, or (if SQL_LOGICAL_CHAR is set to "ON" or "on") by the maximum number of bytes of storage that the largest logical character in the code set of the database locale requires.

For example, if the integer expansion factor is 4, then a CHAR(10) data type specification requests 40 bytes of storage, creating a CHAR(40) data type in standard SQL notation, despite the CHAR(10) declaration.

For NVARCHAR and VARCHAR data types, the declared reserved size, which specifies the minimum storage, is not affected by this feature. For example, with

the same integer setting is 4, then a VARCHAR(10,5) data type specification, with 4 as the expansion factor, requests a maximum of 40 bytes of storage with 5 of these bytes reserved, creating a VARCHAR(40, 5) data type in standard SQL notation, despite the VARCHAR(10,5) declaration. (The reserve size parameters of VARCHAR and NVARCHAR are not affected by the SQL_LOGICAL_CHAR setting, because the minimum size of a multibyte character is 1 byte. In this example, the minimum size of five multibyte characters is 5 bytes, so that declared size remains unchanged.)

When a valid SQL_LOGICAL_CHAR setting greater than 1 is in effect, a VARCHAR or NVARCHAR declaration with no size specification is interpreted as one logical character, and the resulting data type occupies the same number of bytes of storage as the SQL_LOGICAL_CHAR setting.

For LVARCHAR column declarations with no size specified, the default size is interpreted as 2048 logical characters. When LVARCHAR is used in I/O operations on opaque data types, however, the limit on the maximum size is determined by the operation system, and the SQL_LOGICAL_CHAR setting is ignored.

If a client session connects to a database in which the SQL_LOGICAL_CHAR configuration parameter was enabled at the time of database creation, this setting takes effect at connection time. The SQL_LOGICAL_CHAR setting for a database cannot be changed, and persists until the database is dropped, even if the Informix instance that manages the database is stopped and restarted with a new SQL_LOGICAL_CHAR setting.

Whether the SQL_LOGICAL_CHAR configuration parameter is set to enable or disable the expansion of declared storage sizes, its setting specifies how data type declarations are interpreted for all sessions of the Informix instance.

For embedded languages such as ESQL/C, character data type declarations are expanded when they are passed to Informix by the client application.

Automatic resizing of the expansion factor

When SQL_LOGICAL_CHAR is set to a valid digit, and the current session creates a database, IBM Informix compares the SQL_LOGICAL_CHAR value with the maximum number of bytes that any logical character requires in the code set of the database locale.

If the SQL_LOGICAL_CHAR setting is greater than that maximum number of bytes, the database uses the maximum value for the locale as the new expansion factor, overriding what the configuration file specifies. The SQL_LOGICAL_CHAR setting in the configuration file remains unchanged, and continues to act as the default expansion factor for the creation of other databases.

Chapter 4. Database server features

These topics describe how the GLS feature affects the database server.

It covers the following main topics:

- Which operating-system files the database server can access
- When the database server uses code-set conversion
- Which database server utilities provide support for the GLS feature

For more information about these database server features, see the *IBM Informix Administrator's Guide*. For more information about database server utilities, see the *IBM Informix Administrator's Reference*. For information about migrating to a different IBM Informix database server, see the *IBM Informix Migration Guide*.

GLS support by IBM Informix database servers

The database server can perform read and write operations to the operating-system files:

The operating-system files are:

- Diagnostic files

  Diagnostic files include the following files:

  – `af.xxx`
  – `shmem.xxx`
  – `gcore.xxx` (UNIX)
  – core

  The database server generates diagnostic files when you set one or more of the following configuration parameters in UNIX:

  – DUMPDIR
  – DUMPSHMEM
  – DUMPCNT
  – DUMPCORE
  – DUMPGCORE
- Message-log file

  The database server generates a user-specified message-log file when you set the MSGPATH configuration parameter.

These operating-system files reside on the server computer, where the database server resides. When the database server reads from or writes to these files, it must use a code set that the server computer supports. The database server obtains this code set from the server locale.

Set the server locale with the **SERVER_LOCALE** environment variable. If you do not set **SERVER_LOCALE**, the database server uses the default locale, as the server locale. For details, see "The SERVER_LOCALE environment variable" on page 2-21.

To perform code-set conversion and handle non-ASCII characters that are associated with read and write operations on operating-system files, the database

© Copyright IBM Corp. 1996, 2013

server determines the database server code set (the code set that the database server locale supports). For information about the use of non-ASCII characters, see "Non-ASCII characters in identifiers" on page 3-1.

Database server code-set conversion

These topics summarize the code-set conversion that the database server performs.

For more general information about code-set conversion, see "Perform code-set conversion" on page 1-29.

An IBM Informix database server automatically performs code-set conversion between the code sets of the server-processing locale and the server locale when the following conditions are true:

- The **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables are set such that the code sets of the server-processing locale and the server locale are different.
- A valid code-set conversion exists between the code sets of the server-processing locale and server locale.

For a list of files for which IBM Informix database servers perform code-set conversion, see "GLS support by IBM Informix database servers" on page 4-1. For information about GLS code-set conversion files, see "Code-set-conversion files" on page A-8.

Enterprise Replication supports replication between database servers that use different code sets. See Enabling code set conversion between replicates for more information.

After the database server creates the operating-system file, it has generated a file name and written file contents in the code set of the server locale (the server code set). Any IBM Informix product or client application that needs to access this file must have a server-processing locale that supports this same server code set. You must ensure that the appropriate **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables are set so that the server-processing locale supports a code set with these non-ASCII characters. For more information about the server-processing locale, see "Determine the server-processing locale" on page 1-26.

The database server checks the validity of a file name with respect to the server-processing locale before it references the file name.

Data that the database server converts

When the database server transfers data to and from its operating-system files, it handles any differences in the code sets of the server-processing locale and the server locale.

The database server handles these differences as follows:

- If these two code sets are the same, the database server can read from or write to its operating-system files in the code set of the server locale.
- If these two code sets are different and an IBM Informix code-set conversion exists between them, the database server automatically performs code-set conversion when it reads from or writes to its operating-system files.

For code-set conversion to resolve the difference in code sets, the server locale must support the actual code set that the database server used to create the file. For more information, see “Make sure that your product supports the same code set” on page 2-8.

- If these two code sets are different, but no IBM Informix code-set conversion exists, the database server cannot perform code-set conversion.

  If the database server reads from or writes to an operating-system file for which no code-set conversion exists, it uses the code set of the server-processing locale to perform the read or write operation.

Locale-specific support for utilities

This section provides information that is specific to the use of the GLS feature by database server utilities.

For a complete description of utilities, see your *IBM Informix Administrator's Reference*.

For information about database server utilities for auditing, see the *IBM Informix Security Guide*.

Enterprise Replication supports replication between database servers that use different code sets. This functionality is useful for converting servers to the Unicode code set with minimal application downtime, for converting servers from one code set to another, and for replicating data between servers in different locals. You enable replication between code sets by using the **UTF8** option when creating the replicate definition. See Enabling code set conversion between replicates for more information.

Database server utilities and SQL utilities are client applications that request information from an instance of the database server. Therefore, these utilities use the **CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE** environment variables to obtain the name of a nondefault locale, as follows:

- If a database utility is to use a nondefault code set to accept input (including command-line arguments) and to generate output, you must set the **CLIENT_LOCALE** environment variable.
- If a database utility accesses a database with a nondefault locale, you must set the **DB_LOCALE** environment variable.
- If a database utility causes the database server to write data on the server computer that has a nondefault code set, you must set the **SERVER_LOCALE** environment variable.

These utilities also perform code-set conversion if the database and the client locales support convertible code sets. For more information about code-set conversion, see “Perform code-set conversion” on page 1-29.

Changes to locale environment variables should also be reflected in the Windows `registry` database under HKEY_LOCAL_MACHINE.

Non-ASCII characters in database server utilities

Most database server utilities support non-ASCII characters in command-line arguments. These utilities interpret all command-line arguments in the client code set (which **CLIENT_LOCALE** defines).

The following table shows utilities that accept non-ASCII characters in command-line arguments or produce non-ASCII output.

Utility name	Non-ASCII characters in command-line arguments	Non-ASCII output
onaudit	-f *input_file*	Yes
oncheck	-cc -pc *database* -ci -cI -pk -pK -pl -pL *database*:*table#index_name* -ci -cI -pk -pK -pl -pL -cd -cD -pB -pt -pT -pd -pD -pp *database*:*table*	Yes
onload	*database*:*table* -i *old_index new_index* -t *tape_device*	Yes
onlog	-d *tape_device*	
onpload	-d *source* -j *jobname* -p *projectname*	Yes
onshowaudit	-f *input_file* -s *server_name*	Yes
onspaces	-p *pathname* -f *filename*	
onstat	-o *filename -dest* *filename_source*	Yes
onunload	*database*:*table* -t *tape_device*	Yes

Non-ASCII characters in SQL utilities

SQL utilities also accept non-ASCII characters in command-line arguments and generate any output in the client code set.

These SQL utilities are:

- **chkenv**
- **dbexport**
- **dbimport**
- **dbload**
- **dbschema**

For a description of the **chkenv** utility, refer to the *IBM Informix Guide to SQL: Reference*. For a description of the **dbload**, **dbschema**, **dbexport**, and **dbimport** utilities, see the *IBM Informix Migration Guide*. For information about DB-Access, see the *IBM Informix DB-Access User's Guide*.

The DB-Access utility generates labels and messages in the code set of the client locale.

Locale support for C User-defined routines (Informix and DB API)

IBM Informix allows you to create user-defined routines (UDRs) that are written in the C programming language.

These C UDRs use the DataBlade API to communicate with the database server. For a complete description of the DataBlade API, see the *IBM Informix DataBlade API Programmer's Guide*. This section describes how to *globalize* a C UDR.

Globalization is the process of creating a user-defined routine (UDR) that can support different languages, territories, and code sets without changing or recompiling its code.

A globalized C UDR must handle the following GLS considerations:

- Where can the UDR use non-ASCII characters in source code?
- What steps must the C UDR take when copying character data?
- How can the UDR access GLS locales?
- How does the UDR handle code-set conversion?
- How does the UDR handle locale-specific end-user formats?
- How can the UDR access globalized exception messages?
- How can the UDR access globalized tracing messages?
- How do opaque-type support functions handle locale-sensitive data?

Current processing locale for UDRs

To access a database, a client application first requests a connection to the database server, which must verify that it can access the specified database and establish the connection between the client and this database.

In the process, the database server establishes the server-processing locale to use the duration of the connection. When the client application executes a UDR, this UDR executes on the server computer in the context of the server-processing locale. This locale is often called the *current processing locale*.

Many user-defined routines handle non-ASCII data correctly even if they were originally written for ASCII data. Some routines, however, might perform abnormally. To globalize your C UDR, you must ensure that your UDR handles the server-processing locale in any GLS-related operations. If the UDR does not properly support the server-processing locale, the routine might return unexpected results or an error message.

Non-ASCII characters in source code

Non-ASCII characters might appear in the contexts in a C-UDR source file:

These characters might appear in the following statements:

- In C-language statements, such as variable declarations and **if** statements
- In SQL statements, which are sent to the database server through the **mi_exec( )** or **mi_exec_prepared_statement( )** functions

In C-language statements

The C compiler must recognize the code set that you use in your C-language statements.

The capabilities of your C compiler might limit your ability to use non-ASCII characters within the C-language statements in a UDR source file. For example, some C-language compilers support multibyte characters in literals or comments only.

If the C compiler does not fully support non-ASCII characters, it might not successfully compile a UDR that contains these characters. In particular, the following situations might affect compilation of your UDR:

- Multibyte characters might contain C-language tokens.

  A component of a multibyte character might be indistinguishable from certain single-byte characters such as percent (%), comma, backslash (\ ), and double quotation mark ("). If such characters exist in a quoted string, the C compiler might interpret them as C-language tokens, which can result in compilation errors or even lost characters.
- The C compiler might not be 8-bit clean.

  If a code set contains non-ASCII characters (with code values that are greater than 127), the C compiler must be 8-bit clean to interpret the characters. To be 8-bit clean, a compiler must read the eighth bit as part of the code value; it must not ignore or put its own interpretation on the meaning of this eighth bit.

Tip: The C compiler must also recognize the ASCII code set to be able to interpret the names of the DataBlade API functions within your C UDR.

In SQL statements

In C UDRs, SQL statements occur as literal strings to the **mi_exec( )** and **mi_prepare( )** functions.

The C compiler does not parse these literal strings. Therefore, it does not need to recognize the code set of the characters in these SQL statements.

Within a C source file, you can use non-ASCII characters in SQL statements for the following objects:

- Names of SQL identifiers such as databases, tables, columns, views, constraints, prepared statements, and cursors

  For more information, see "Name database objects" on page 3-1.
- Literal strings

  For example, in a UDR, the following use of multibyte characters is valid:

  ```
  mi_exec(conn,
     "insert into tbl1 (nchr1) values 'A¹A²B¹B²'", 0);
  ```
- File names and path names, as long as your operating system supports multibyte characters in file names and path names

Important: To use non-ASCII characters in your SQL statements, your server-processing locale must include either a code set that supports these characters or a code set that is compatible with the character code set. For information about how to perform code-set conversion, see "Character strings in UDRs" on page 4-8.

Copy character data

When you copy data, you must ensure that the buffers are an adequate size to hold the data.

If the destination buffer is not large enough for the multibyte data in the source buffer, the data might be truncated during the copy. For example, the following C code fragment copies the multibyte data $A^1A^2A^3B^1B^2B^3$ from **buf1** to **buf2**:

```
char buf1[20], buf2[5];
...
stcopy("A1A2A3B1B2B3", buf1);
...
stcopy(buf1, buf2);
```

Because **buf2** is not large enough to hold the multibyte string, the copy truncates the string to $A^1A^2A^3B^1B^2$. To prevent this situation, ensure that the multibyte string fits into a buffer before the DataBlade API module performs the copy.

The IBM Informix GLS library

The IBM Informix GLS library is an application programming interface (API) through which developers of user-defined routines and of DataBlade modules can create globalized applications.

Character processing with IBM Informix GLS

The macros and functions of IBM Informix GLS provide access within a DataBlade API module to GLS locales for culture-specific information.

This library contains functions that provide the following capabilities:

- Process single-byte and multibyte characters
- Format date, time, and numeric data to locale-specific formats

Compatibility of wide-character data types

Wide character data types are an alternative form for the processing of multibyte characters. A wide-character form of a code set involves the normalization of the size of each multibyte character so that each character is the same size.

A legacy DataBlade API module might use any of the following data types to hold wide characters.

Wide-character data type	Description	Drawback
mi_wchar	A legacy DataBlade API data type currently defined as unsigned short on all systems	The DataBlade API does not provide wide-character functions that operate on **mi_wchar** values.
wchar_t	An operating-system data type that is platform-specific	The operating-system provides wide-character functions that operate on **wchar_t** values. Use of these functions is platform-specific.

The IBM Informix GLS library provides the **gl_wchar_t** data type for support of wide characters. IBM Informix GLS also provides its own set of wide-character functions that operate on **gl_wchar_t**. Use of the IBM Informix GLS wide-character functions removes platform dependency from your application and provides access within your DataBlade API module to IBM Informix GLS locales.

The IBM Informix GLS library does not provide any functions for conversion between **gl_wchar_t** and **mi_wchar** or **gl_wchar_t** an **wchar_t**. If a DataBlade API module continues to use either **mi_wchar** or **wchar_t** and also needs to use the IBM Informix GLS wide-character processing, you must write code to perform any necessary conversions.

Code-set conversion and the DataBlade API

Within a UDR, the DataBlade API does not perform any code-set conversion automatically.

Your C UDR might need to perform code-set conversion in the following situations:

- In strings that contain SQL statements
- In an opaque-type support function for an opaque type that contains character data

Character strings in UDRs

When your C UDR contains character strings that are sent to the database server, it must perform any required code-set conversion on these strings.

This code-set conversion must handle any differences between the code set of this character string and the code set of the server-processing locale in which the UDR executes.

For example, the DataBlade API does not perform code-set conversion on the multibyte table name, $A^1A^2A^3B^1B^2$, in the following SELECT statement:

```
mi_exec(conn, "SELECT * from A1A2A3B1B2", 0);
```

If your UDR might execute in a server-processing locale that does not include a code set that supports characters in your SQL statements, the UDR can explicitly perform code-set conversion between the code sets of the server-processing locale and a specified locale.

Character strings in opaque-type support functions

The client application performs code-set conversion of non-opaque-type data that is transferred to and from the client, but the database server does not know about the internal format of an opaque data type.

Therefore, for opaque data types, the support functions are responsible for explicitly converting any string that is not in the code set of the server-processing locale.

You might need to perform code-set conversion in the following opaque-type support functions:

- In the input and output support functions: to convert the external format of the opaque type between the code sets of the client locale and the server-processing-locale
- In the receive and send support functions: to convert any character fields in the internal structure of the opaque type

Tip: The code that the Informix DataBlade Developers Kit (DBDK) generates for opaque-type input and output support functions handles external formats from nondefault locales.

The DataBlade API provides the following functions for code-set conversion in the support functions of an opaque data type.

Code-set conversion on an opaque type	DataBlade API function
Perform code-set conversion on a string argument from the code set of the server-processing locale to that of the client locale	**mi_put_string( )**
Perform code-set conversion on a string from the code set of the client locale to that of the server-processing locale	**mi_get_string( )**

For more information about the syntax of these DataBlade API functions, see the function reference in the *IBM Informix DataBlade API Programmer's Guide*.

Locale-specific data formatting

When a C UDR handles strings that contain end-user formats for date, time, numeric, or monetary data, you must write the UDR so that it handles any locale-specific formats of these end-user formats.

The DataBlade API provides functions that convert between the internal representation of several data types and its end-user format.

The following DataBlade API functions convert an internal database value to a string that uses the locale-specific end-user format.

DataBlade API function	Description
mi_date_to_string()	Uses the locale-specific end-user date format to convert an internal DATE value to its string equivalent.
mi_money_to_string()	Uses the locale-specific end-user monetary format to convert an internal MONEY value to its string equivalent.
mi_decimal_to_string()	Uses the locale-specific end-user numeric format to convert an internal DECIMAL value to its string equivalent.

Important: The **mi_datetime_to_string()** and **mi_interval_to_string()** functions do not format strings in the date and time formats of the current processing locale. Instead, they create a date, time, or interval string in a fixed ANSI SQL format.

The following DataBlade API functions interpret a string in its locale-specific end-user format and convert it to its internal database value.

DataBlade API function	Description
mi_string_to_date()	Converts a string in its locale-specific date end-user format to its internal DATE format.
mi_string_to_money()	Converts a string in its locale-specific currency end-user format to its internal MONEY format.
mi_string_to_decimal()	Converts a string in its locale-specific numeric end-user format to its internal DECIMAL format.

Important: The **mi_string_to_datetime()** and **mi_string_to_interval()** functions do not interpret the date and time formats of the current processing locale. Instead, they interpret the date/time or interval string in a fixed ANSI SQL format.

Globalized exception messages

The DataBlade API function **mi_db_error_raise()** sends an exception message to an exception callback.

This message can be either of the following:

- A *literal message*, which you provide as the third argument to **mi_db_error_raise()**
- A *customized message* that is associated with a value of **SQLSTATE**, which you provide as the third argument to **mi_db_error_raise()**

  The **mi_db_error_raise()** function can raise exceptions with customized messages, which DataBlade modules and UDRs can store in the **syserrors** system catalog table. The **syserrors** table maps these messages to five-character **SQLSTATE** values. In the **syserrors** table, you can associate a locale with the text of a customized message.

For general information about how to specify a literal message in **mi_db_error_raise()** and how to specify a customized message for **mi_db_error_raise()**, see the topics on how to handle exceptions and events in the *IBM Informix DataBlade API Programmer's Guide*.

This section describes the following tasks about how to raise locale-specific exception messages:

- How to add a locale-specific exception message to the **syserrors** system catalog table
- How the choice of locale in a customized message affects the way that **mi_db_error_raise()** searches for a customized message
- How to specify parameter markers that contain non-ASCII characters

Insert customized exception messages

You can store customized status codes and their associated messages in the **syserrors** system catalog table.

To create a customized exception message, insert a row directly in the **syserrors** table. The **syserrors** table provides the following columns for a globalized exception message.

Column name	Description

sqlstate	The **SQLSTATE** value that is associated with the exception You can use the following query to determine the current list of **SQLSTATE** message strings in **syserrors**:
	`SELECT sqlstate, locale, message`
	`FROM syserrors`
	`ORDER BY sqlstate, locale`
	
	For more information about how to determine **SQLSTATE** values, see the *IBM Informix DataBlade API Programmer's Guide*.

message	The text of the exception message, with characters in the code set of the target locale By convention, do not include any newline characters in the message.
locale	The locale with which the exception message is to be used The **locale** column identifies the language and code set used for the globalization of error and warning messages. This name is the name of the target locale of the message text.

For more information about the **syserrors** system catalog table, see the topics that describe the system catalog in the *IBM Informix Guide to SQL: Reference*. Do not allow any code-set conversion when you insert the text in **syserrors**.

If the code sets of the client and database locales differ, temporarily set both the **CLIENT_LOCALE** and **DB_LOCALE** environment variables in the client environment to the name of the database locale. This workaround prevents the client application from performing code-set conversion.

If you specify any parameters in the message text, include only ASCII characters in the parameter names, so that the parameter name can be the same for all locales. Most code sets include the ASCII characters. For example, the following INSERT statements insert new messages in **syserrors** whose **SQLSTATE** value is "03I01":

```
INSERT INTO syserrors VALUES ("03I01", "en_us.8859-1", 0, 1,
      "Operation Interrupted.")
INSERT INTO syserrors VALUES ("03I01", "fr_ca.8859-1", 0, 1,
      "Traitement Interrompu.")
```

The '03I01' **SQLSTATE** value now has two locale-specific messages. The database server chooses the appropriate message based on the server-processing locale of the UDR when it executes. For more information about how **mi_db_error_raise( )** locates an exception message, see "Search for customized messages" on page 4-12.

Insert a localized exception message from a C UDR

As noted in the previous section, when you create messages for exceptions raised within user-defined routines (UDRs) by **mi_db_error_raise( )**, the locale of the message text must match the server-processing locale. If these locales are different, use of an SQL script or of a C UDR that calls the **mi_exec( )** function to insert the message is not reliable, because the SQL parser issues an exception when it encounters characters that it does not recognize.

To avoid this restriction, you can use a UDR that prepares the INSERT statement (with **mi_prepare( )**) to load the error messages:

- Use placeholders ('?' symbols) for the **SQLSTATE** value and the error-message text. These values are in the first (**sqlstate**) and last columns (**message**) of the **syserrors** system catalog table.
- Hardcode the name of the locale that the message text uses. The locale name is in the second column (**locale**) of **syserrors**.

For example, the following line prepares an INSERT statement for messages in the default locale (**en_us**) on a UNIX system:

```
stmt = mi_prepare(conn,
   "insert into syserrors (?, 'en_us.8859-1', 0, 1, ?)", NULL);
```

When executing this statement, you must provide values for the placeholders (**sqlstate** and **message**) and then use the **mi_exec_prepared_statement( )** function to send the prepared INSERT statement to the database server.

The following UDR code uses a message array (**enus_msg**) to hold the SQLSTATE values and their associated message text. It puts information about each element of this message array in the appropriate placeholder arrays (**args**, **lens**, **nulls**, and **types**) of the **mi_exec_prepared_statement( )** function.

```
#include <stdio.h>
#include <string.h>
#include "mi.h"

#define MAX_MSG 3
char *enus_msg[MAX_MSG][2] =  {
   "XT010", "First error message for insertion",
   "XT020", "Second error message for insertion",
   "XT030", "Third error message for insertion"
```

```
    };

/*
 * Title: gls_insert_enus
 * Purpose: Add localized messages to 'syserrors' system error table
 *          for given locale, independent of session locale setting.
 */
mi_integer
gls_insert_enus()
{
MI_DATUM       args[2];             /* pointers to column values */
mi_integer     lens[2];             /* lengths of column values */
mi_integer     nulls[2];            /* null capability of columns */
mi_string     *types[2];            /* types of columns */
mi_integer     i;
MI_STATEMENT  *stmt;
MI_CONNECTION *conn = mi_open(NULL, NULL, NULL);

/*
 * Prepare statement using placeholder values for sqlstate and message
 * columns and fixed values for locale, level, and seqno columns.
 */
stmt = mi_prepare(conn,
         "insert into syserrors values(?,'en_us.8859-1',0,1,?)", NULL);
for (i=0; i<MAX_MSG; i++)           /* Loop through message array */
{
   args[0] = (MI_DATUM)enus_msg[i][0];
      /* Set pointer to sqlstate string */
   lens[0] = strlen(args[0]);      /* Set length of sqlstate string */
   nulls[0] = MI_FALSE;            /* Set null handling capability */
   types[0] = "char(5)";           /* Set sqlstate column type */
   args[1] = (MI_DATUM)enus_msg[i][1];
      /* Set pointer to message string */
   lens[1] = strlen(args[1]);      /* Set length of message string */
   nulls[1] = MI_FALSE;            /* Set null handling capability */
   types[1] = "varchar(255)";      /* Set message column type */

mi_exec_prepared_statement(stmt,0,0,2,args,lens,nulls,types,
   NULL,NULL);
}
mi_close(conn);
return 0;
}
```

For descriptions of executing prepared statements and of how to add customized messages to the **syserrors** system catalog table, see the *IBM Informix DataBlade API Programmer's Guide*.

Search for customized messages

When the **mi_db_error_raise( )** function initiates a search of the **syserrors** system catalog table, it requests the message in which all components of the locale (language, territory, code set, and optional modifier) are the same in the current processing locale and the **locale** column of **syserrors**.

For C UDRs that use the default locale, the current processing locale is U.S. English (**en_us**). When the current processing locale is U.S. English, **mi_db_error_raise( )** looks only for messages that use the U.S. English locale. For C UDRs that use nondefault locales, however, the current processing locale is the server-processing locale.

For a description of how **mi_db_error_raise( )** searches for messages in the **syserrors** system catalog table, see the chapter on exceptions in the *IBM Informix DataBlade API Programmer's Guide*.

Specify parameter markers

The customized message in the **syserrors** system catalog table can contain *parameter markers*. These parameter markers are strings of characters enclosed by a single percent (%) symbol on each end (for example, %TOKEN%).

A parameter marker is treated as a variable for which the **mi_db_error_raise()** function can supply a value. The **mi_db_error_raise()** function assumes that any message text or message parameter strings that you supply are in the server-processing locale. For a complete description of how to specify parameter markers for a customized message, see the *IBM Informix DataBlade API Programmer's Guide*.

Globalized tracing messages

The API supports trace messages that correspond to a particular locale. The current database locale determines which code set the trace message uses.

Based on the current database locale, a given tracepoint can produce a globalized trace message. Globalized tracing enables you to develop and test the same code in many different locales.

To provide globalized tracing support, the API provides the following capabilities:

- The **systracemsgs** system catalog table stores globalized trace messages.
- Two globalized trace functions, **gl_dprintf( )** and **gl_tprintf( )**, format globalized trace messages.

Insert messages in the systracemsgs system catalog table

The **systracemsgs** system catalog table stores globalized trace messages that you can use to debug your C UDRs.

To create a globalized trace message, insert a row directly into the **systracemsgs** table.

The **systracemsgs** table describes each globalized trace message.

Column name	Description
name	The name of the trace message
locale	The locale with which the trace message is to be used
message	The text of the trace message

The combination of message name and locale must be unique within the table. Once you insert a new trace class into **systracemsgs**, the database server assigns it a unique identifier, called a *trace-message identifier*. It stores the trace-class identifier in the **msgid** column of **systracemsgs**. Once a trace message exists in the **systracemsgs** table, you can specify the message either by name or by trace-message identifier to API tracing functions.

The trace-message text can be a string of text in the appropriate language and code set for the locale, and can contain *tokens* to indicate where to substitute a piece of text. Token names are delimited between percent (%) symbols. The following INSERT statement puts a new message called **qp1_exit** in the **systracemsgs** table:

```
INSERT INTO informix.systracemsgs(name, locale, message)
VALUES ('qp1_exit', 'en_us.8859-1',
   'Exiting msg number was  the input is still %i%')
```

This message text is in English and therefore the **systracemsgs** row specifies the default locale of U.S. English.

This second message is the French version of the **qp1_exit** message and therefore the **systracemsgs** row specifies a French locale on a UNIX system (**fr_fr.8859-1**):

```
INSERT INTO informix.systracemsgs(name, locale, message)
VALUES ('qp1_exit', 'fr_fr.8859-1',
   'Le numéro de message en sortie était  \
   l'entrée est toujours %i%')
```

Enter message text in the language of the server locale, with any characters available in the server code set. To insert a variable, enclose the variable name with a single percent sign on each end (for example, %a%). When the database server prepares the trace message for output, it replaces each variable with its actual value.

Put globalized trace messages into code

The DataBlade API provides the tracing functions to insert globalized tracepoints into UDR code.

The following tracing functions can be used to insert globalized tracepoints into UDR code:

- The GL_DPRINTF macro formats a globalized trace message and specifies the threshold for the tracepoint. The syntax for GL_DPRINTF is as follows:

  ```
  GL_DPRINTF(trace_class, threshold,
     (message_name [,toktype, val]...,MI_LIST_END));
  ```

- The **gl_tprintf( )** function formats a globalized trace message but does not specify a tracepoint threshold.

  The **gl_tprintf( )** function is for use within a trace block, which uses the **tf( )** function to compare a specified threshold with the current trace level. The syntax for **gl_tprintf( )** is as follows:

  ```
  gl_tprintf(message_name [,toktype ,val]...,
     MI_LIST_END);
  ```

Syntax elements for both GL_DPRINTF and **gl_tprintf( )** have these values:

trace_class
: Either a trace-class name or the trace-class identifier integer value expressed as a character string.

threshold
: A nonnegative integer that sets the tracepoint threshold for execution.

message_name
: The identifier for a globalized message stored in the **systracemsgs** system catalog table of the database.

toktype
: A string made up of a token name followed by a single percent (%) symbol followed by a single character output specifier as used in **printf** formats.

val
: A value expression to be output that must match the type of the output specifier in the preceding token.

MI_LIST_END
: A macro constant that ends the variable-length list.

  Important: The MI_LIST_END constant marks the end of the variable-length list. If you do not include MI_LIST_END, the user-defined routine might fail.

This globalized trace statement uses the GL_DPRINTF macro:

```
i = 6;
/* If the current trace level of the funcEntry class is greater
 * than or equal to 20, find the version of the qp1_entry
 * message whose locale matches the current database locale
 */
GL_DPRINTF("funcEntry", 20,
         ("qp1_entry",
         "ident%s", "one",
         "i%d", i,
         MI_LIST_END));
```

In the default locale, if the current trace level of the **funcEntry** class is greater than or equal to 20, this tracepoint generates the following trace message:

```
13:21:51  Exiting msg number was one; the input is still 6
```

The following globalized trace block that uses the **gl_tprinf()** function:

```
i = 6;
/* Compare current trace level of "funcEnd" class and
 * with a tracepoint threshold of 25. Continue execution of
 * trace block if trace level >= 25
 */
if ( tf("funcEnd", 25) )
   {
      i = doSomething();
   /* Generate an internationalized trace message (based
    * on current database locale) */
      gl_tprintf("qp1_exit", "ident%s", "deux", "i%d", i,
      MI_LIST_END);
   }
```

If the locale is French and the current trace level of the **funcEntry** class is greater than or equal to 25, the tracepoint generates this trace message:

```
13:21:53 Le numéro de message en sortie était deux; l'entrée
est toujours 6
```

The database server writes the trace messages in the trace-output file in the code set of the locale associated with the message. If the trace message originated from the **systracemsgs** system catalog table, its characters are in the code set of the locale specified in the **locale** column of its **systracemsgs** entry. The database server might have performed code-set conversion on these trace messages if the code set in the UDR source is different from (but compatible with) the code set of the server-processing locale.

Search for trace messages

To write a globalized trace message to your trace-output file, the database server must locate a row in the **systracemsgs** system catalog table whose **locale** column matches (or is compatible with) the server-processing locale for your UDR.

Therefore, to see a particular trace message in the trace-output file, environment variables that specify the locale (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**) must be set so that the database server generates a server-processing locale that matches an entry in the **systracemsgs** system catalog table.

The database server searches the **systracemsgs** table for an entry with the same name as the tracepoint and a locale in which all components of the locale (language, territory, and code set) are the same in the current processing locale and the **locale** column of **systracemsgs**. If only the language and territory match, the database server converts the code set. If no message has matching language and

territory, it uses the first available message with the correct language. If there is no message in the appropriate language, it uses the message for the default language, **en_us**.

Locale-sensitive data in an opaque data type

An opaque data type is fully encapsulated. Its internal structure is not known to the database server.

The database server cannot automatically perform the locale-specific tasks such as code-set conversion on character data or locale-specific formatting of date, numeric, or monetary data. When you create an opaque data type, you must write the support functions of the opaque type so that they handle any locale-sensitive data.

In particular, consider how to handle any locale-sensitive data when you write the following support functions:

- The **input( )** and **output( )** support functions
- The **receive( )** and **send( )** support functions

The DataBlade API and IBM Informix GLS provide GLS support for opaque-type support functions written in C. The following sections summarize GLS considerations for these support functions. For general information about the support functions of an opaque data type, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Globalized input and output support functions

The **input( )** and **output( )** support functions convert the opaque data type from its internal to an external representation, and vice versa.

The internal representation of an opaque data type is the C structure that stores the opaque-type data. Each opaque type also has a character-based format, known as its *external representation*, which is received by the database server as an LVARCHAR value. This can hold single-byte (ASCII and non-ASCII) and multibyte character strings, depending on the locale of the client application. (The data length of an LVARCHAR external representation is limited only by the operating system, not by the 32,739 byte maximum size of LVARCHAR columns in IBM Informix databases.)

Client applications perform code-set conversion on LVARCHAR data types. The ability to transfer the data between a client application and database server, however, is not sufficient to support locale-sensitive data in opaque data types. It does not ensure that data values are correctly manipulated at the destination.

The **input()** and **output()** support functions convert the opaque data type as follows:

- The **input()** function converts the external representation of the data type to the internal representation.
- The **output()** function converts the internal representation of the data type to the external representation.

Opaque-type support functions written as C UDRs must ensure that these functions correctly handle any locale-sensitive data, including these tasks.

Locale-sensitive task	For more information
Any code-set conversion on character data	"Code-set conversion and the DataBlade API" on page 4-8
Any handling of multibyte or wide characters in character data	"The IBM Informix GLS library" on page 4-7
Any formatting of locale-specific date, numeric, or monetary data	"Locale-specific data formatting" on page 4-9

Globalized send and receive support functions

The **send( )** and **receive( )** functions support binary transfer of opaque data types.

That is, they convert the opaque data type from its internal representation on the client computer to its internal representation on the server computer (where it is stored), as follows:

- The **receive( )** function converts the internal representation of the opaque data type on the client computer to its internal representation on the server computer.
- The **send( )** function converts the internal representation of the opaque data type on the server computer to its internal representation on the client computer.

If the internal representation contains character data, the client application cannot perform any locale-specific translations, including these.

Locale-sensitive task	For more information
Any code-set conversion on character data	"Character strings in opaque-type support functions" on page 4-8
Any handling of multibyte or wide characters in character data	"The IBM Informix GLS library" on page 4-7

When you write **receive( )** and **send( )** support functions as C UDRs, you must ensure that these functions handle these locale-sensitive tasks correctly.

Chapter 5. General SQL API features (ESQL/C)

These topics explain how the GLS feature affects applications that you develop with the IBM Informix Client Software Development Kit.

Support for GLS in IBM Informix client applications

To connect to a database, an IBM Informix ESQL/C client application requests a connection from the database server. The database server must verify that it can access the database and establish the connection between the client and the database.

Your client application performs the following tasks:

- Sends its client and database locale information to the database server

  The Informix ESQL/C program performs this step automatically when it requests a connection.
- Checks for connection warnings that the database server generates

  You must include code in your Informix ESQL/C program to perform this step.

Client application code-set conversion

These topics summarize the code-set conversion that a client product performs.

For more general information about code-set conversion, see "Perform code-set conversion" on page 1-29.

The client application automatically performs code-set conversion between the client and database code sets when both of these conditions are true:

- The code sets of the client and database locales do not match.
- A valid object code-set conversion exists for the conversion between the client and database code sets.

When the client application begins execution, it compares the names of the client and database locales to determine whether to perform code-set conversion. If **CLIENT_LOCALE** is not set, the client application assumes that the client locale is the default locale. If **DB_LOCALE** is not set, the client application assumes that the database locale is the same as the client locale (the value of the **CLIENT_LOCALE** setting).

If the client and database code sets are the same, no code-set conversion is needed. If the code sets do not match, however, the client application must determine whether the two code sets are *convertible*. Two code sets are convertible if the client can locate the associated code-set-conversion files. These code-set-conversion files must exist on the client computer.

On UNIX, you can use the **glfiles** utility to obtain a list of code-set conversions that your Informix product supports. For more information, see "The glfiles utility (UNIX)" on page A-12. On Windows, you can examine the directory `%INFORMIXDIR%\gls\cvY` to determine the GLS code-set conversions that your Informix product supports. For more information about this directory, see "Code-set-conversion files" on page A-8.

© Copyright IBM Corp. 1996, 2013

If no code-set-conversion files exist, the client application generates a run time error when it starts to indicate incompatible code sets. If code-set-conversion files exist, the client application automatically performs code-set conversion when it sends data to or receives data from the database server.

When a client application performs code-set conversion, it assumes that:

- All data values that are processes are handled in the client code set.
- All databases that the client application accesses on a single database server use the same database locale, territory, and code set. When the client application opens a different database, it does not recheck the database locale to determine if the code set has changed.

Important: Check the eighth character field of the SQLWARN array for a warning flag after each request for a connection. If the two database locales do not match, the client application might be performing code-set conversion incorrectly. The client application continues to perform any code-set conversion based on the code set that **DB_LOCALE** supports. If you proceed with such a connection, it is your responsibility to understand the format of the data that is being exchanged.

For example, suppose your client application has **CLIENT_LOCALE** set to **en_us.1252** and **DB_LOCALE** set to **en_us.8859-1**. The client application determines that it must perform code-set conversion between the Windows Code Page 1252 (in the client locale) and the ISO8859-1 code set (in the database locale). The client application then opens a database with the French **fr_fr.8859-1** locale. The database server sets the eighth character field of the SQLWARN array to `W` because the languages and territories of the two locales are different. The database server then uses the locale of the database (**fr_fr.8859-1**) for the localized order of the data.

Your application, however, might use this connection. It might be acceptable for the application to receive the NCHAR and NVARCHAR data that is sorted in a French localized order. Any code-set conversion that the client application performs is still valid because both database locales support the default ISO8859-1 code set.

Instead, if the application opens a database with the Japanese SJIS (**ja_jp.sjis**) locale, the database server sets the SQLWARN warning flag because the language, territory, *and* code sets differ. The database server then uses the **ja_jp.sjis** locale for the localized order of the data.

Your application would probably *not* continue with this connection. When the client application started, it determined that code-set conversion was required between the Windows Code Page 1252 and ISO8859-1 code set. The client application performs this code-set conversion until it terminates.

When you open a database with **ja_jp.sjis**, the client application would perform code-set conversion incorrectly because the code sets are different. It would continue to convert between Windows Code Page 1252 and ISO8859-1 instead of between Windows Code Page 1252 and Japanese SJIS. This situation could lead to corruption of data.

Tip: If your ESQL/C client application uses code-set conversion, you might need to take special programming steps. For more information, see "Handle code-set conversion" on page 6-14.

Data that a client application converts

When the code sets of two locales differ, an IBM Informix client product must use code-set conversion to prevent data corruption of character data.

Code-set conversion converts the following character data elements:

- Values of SQL data types:
  - CHAR, VARCHAR, NCHAR, and NVARCHAR
  - TEXT (the BYTE data type is not converted)
  - LVARCHAR
  - Character data in opaque data types (if their support functions perform the code-set conversions)
- Values of Informix ESQL/C character types (**char**, **fixchar**, **string**, and **varchar**)
- SQL statements, both static and dynamic
- SQL identifiers. These include names of columns, tables, views, prepared statements, cursors, constraints, indexes, triggers, and other database objects. For a list of SQL objects that can include non-ASCII characters in their identifiers, see "Non-ASCII characters in identifiers" on page 3-1.
- SPL text
- Command text
- Error message text in the **sqlca.sqlerrm** field

Globalize client applications

To globalize or localize a client application, use IBM Informix GLS, an application programming interface (API) for applications that use a C-language interface.

For more information, see "GLS support by IBM Informix products" on page 1-5.

Globalization

Globalization is the process of creating or modifying an application so that it can use the correct GLS locale to support different languages, territories, and code sets without changing or recompiling the code.

This process makes IBM Informix database applications easily adaptable to any culture and language. For a database application, you perform globalization on the application that accesses a database, not on the database. The data in a database that the application accesses should already be in a language that the user can understand.

To globalize a database application, design the application so that the tasks in the following table do not make any assumptions about the language, territory, and code set that the application uses at run time.

Application Task	Description
User interfaces	Includes any text that is visible to users, including menus, buttons, prompts, help text, status messages, error messages, and graphics

Application Task	Description
Character processing	Includes the following processing tasks: • Character classification • Character case conversion • Collation and sorting • Character versus byte processing • String traversal • Code-set conversion
Data formatting	Includes any culture-specific formats for numeric, monetary, date, and time values
Documentation	Includes any explanatory material such as printed manuals, online documentation, and readme files
Debugging via tracing (Informix, DB API)	The DataBlade API provides the application or DataBlade developer the capability of using globalized trace messages. It uses in-line code working with system catalog tables: **systracemsgs** and **systraceclasses**. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

A globalized application dynamically obtains language-specific information for these application tasks. Therefore, one executable file for the application can support multiple languages.

Localization

Localization is the process of adapting a product to a specific cultural environment.

This process usually involves the following tasks:

- Creating culture-specific resource files
- Translating message or resource files
- Setting date, time, and money formats
- Translating the product user interface

Localization might also include the translation and production of end-user documentation, packaging, and collateral materials.

To localize a database application, you create a database application for a specific language, territory, and code set. Localization involves the following tasks:

- Ensure that GLS locales exist for the language, territory, and code set you want.
- Translate the character strings in any external resource or message files that the application uses.

Important: A globalized application is much easier localize than a non-globalized application.

Choose a GLS locale

To localize your application, choose a locale that provides the culture-specific information for the language, territory, and code set that the application is to support.

For information about locales, see "Set a GLS locale" on page 1-16.

A globalized application makes no assumptions about how these locales are set at run time. Once the application environment specifies the locales to use, the application can access the appropriate GLS locale files for locale-specific information. As long as a GLS locale is provided that supports a particular language, territory, and code set, the application can obtain the locale-specific information dynamically.

The *current processing locale* (sometimes called just the *current locale*) is the locale that is currently in effect for an application. It is based on one of the following environments:

- The client environment

  IBM Informix ESQL/C creates client applications. Therefore, the current processing locale for Informix ESQL/C applications is the client locale.
- The database that the database server is currently accessing

The current processing locale for DataBlade client applications is the client locale. The current processing locale for DataBlade UDRs is the server-processing locale, which the database server determines from the client, database, and server locales.

Translate messages

A globalized application should not have any language-specific text within the application code.

This language-specific text includes the following kinds of strings:

- Strings that the application displays or writes

  Examples include error messages, informational messages, menu items, and button labels.
- Strings that the application uses internally

  Examples include constants, file names, and literal characters or strings.
- Strings that an user is expected to enter

  Examples include `yes` and `no` responses.

Tip: You do not need to put SQL keywords (such as SELECT, WHERE, INSERT, and CREATE) in a message file. In addition, language keywords (such as **if**, **switch**, **for**, and **char**) do not need to appear in a message file.

In a globalized application, these strings appear as references to external files, called *resource files* or *message files*. To localize these strings of the database application, you must perform the following tasks:

- Translate all strings within the external files.

  The new external files contain the translated versions of the strings that the application uses.
- Set the **DBLANG** environment variable to the subdirectory within **INFORMIXDIR** that contains the translated message files that the IBM Informix products use.

  The **INFORMIXDIR** environment variable indicates the location where the Informix products are installed. You can use the **rgetmsg( )** and **rgetlmsg( )** functions to obtain Informix product messages. For more information about these functions, see the *IBM Informix ESQL/C Programmer's Manual*.

Handle locale-specific data

Each IBM Informix SQL API product contains a processor to process an Informix ESQL/C source file that has embedded SQL and preprocessor statements.

The Informix ESQL/C processor, **esql**, processes C source files.

The processors for Informix ESQL/C products use operating-system files in the following situations:

- They write language-specific source files (`.c`) when they process an Informix ESQL/C source file.

  The Informix ESQL/C processors use the client code set (that the client locale specifies) to generate the file names for these language-specific files.
- They read Informix ESQL/C source files (`.ec`) that the user creates.

  The Informix ESQL/C processors use the client code set to interpret the contents of these Informix ESQL/C source files.

Use the **CLIENT_LOCALE** environment variable to specify the client locale.

Process characters

A GLS locale supports a specific code set, which can contain single-byte characters and multibyte characters.

When your application processes only single-byte characters, it can perform string-processing tasks based on the assumption that the number of bytes in a buffer equals the number of characters that the buffer can hold. For single-byte code sets, you can rely on the built-in scaling for array allocation and access that the C compiler provides.

If your application processes multibyte characters, however, it can no longer assume that the number of bytes in a buffer equals the number of characters in the buffer. Because of the potential of varying number of bytes for each character, you can no longer rely on the C compiler to perform character-processing tasks such as traversing a multibyte-character string and allocating sufficient space in memory for a multibyte-character string.

You can use functions from the IBM Informix GLS library to communicate to your application how to perform globalization on character-processing tasks.

Character-processing tasks include the following:

- String traversal
- String processing
- Character classification
- Case conversion
- Character comparison and sorting

Format data

When you globalize an application, consider how to handle the format of locale-specific data.

The format in which numeric, monetary, and date and time data appears to the user is locale-specific. The GLS locale file defines locale-specific formats for each of these types of data, as the following table shows.

Type of data	Locale-file category
Numeric	NUMERIC
Monetary	MONETARY
Date and Time	TIME

The IBM Informix GLS library provides functions that allow you to perform the following tasks on locale-specific data:

- *Conversion* changes a string that contains locale-specific format to the internal representation of its value.

  You usually perform conversion on a locale-specific string to prepare it for storage in a program variable or a database column.
- *Formatting* changes the internal representation of a value to locale-specific string.

  You usually perform formatting of a locale-specific string to prepare the internal representation of a value for display to the user.

Avoid partial characters

When you use a locale that supports a multibyte code set, make sure that you define buffers large enough to avoid the generation of partial characters.

Possible areas for consideration are as follows:

- When you copy data from one buffer to another
- When you have character data that might undergo code-set conversion

For more detailed examples of partial characters, see "Partial characters in column substrings" on page 3-15.

Copy character data

When you copy data, you must ensure that the buffers are an adequate size to hold the data. If the destination buffer is not large enough for the multibyte data in the source buffer, the data might be truncated during the copy.

For example, the following IBM Informix ESQL/C code fragment copies the multibyte data **A1A2A3B1B2B3** from **buf1** to **buf2**:

```
char buf1[20], buf2[5];
...
stcopy("A1A2A3B1B2B3", buf1);
...
stcopy(buf1, buf2);
```

Because **buf2** is not large enough to hold the multibyte string, the copy truncates the string to **A1A2A3B1B2**. To prevent this situation, ensure that the multibyte string fits into a buffer before the Informix ESQL/C program performs the copy.

Code-set conversion

If you have a character buffer to hold character data from a database, you must ensure that this buffer is large enough to accommodate any expansion that might occur if the application uses code-set conversion. If the client and database locales are different and convertible, the application might need to expand this value.

For more information, see "Perform code-set conversion" on page 1-29.

For example, if the **fname** column is defined as CHAR(8), the following IBM Informix ESQL/C code fragment selects an 8-byte character value into the 10-byte **buf1** host variable:

```
char buf1[10];
...
EXEC SQL select fname into :buf1 from tab1
         where cust_num = 29;
```

You might expect a 10-byte buffer to be adequate to hold an 8-byte character value from the database. If the client application expands this value to 12 bytes, however, the value no longer fits in the **buf1** buffer. The **fname** value is truncated to fit in **buf1**, possibly creating partial characters if **fname** contains multibyte characters. For more information, see "Partial characters in column substrings" on page 3-15.

To avoid this situation, define buffers to handle the maximum character-expansion possible, 4 bytes, in the conversion between your client and database code sets.

Chapter 6. IBM Informix ESQL/C features

These topics explain how the GLS feature affects IBM Informix ESQL/C, an SQL application programming interface (API).

These topics also cover GLS information that is specific to Informix ESQL/C. For additional GLS information for Informix ESQL/C, see Chapter 5, "General SQL API features (ESQL/C)," on page 5-1.

Tip: For features that are not unique to the GLS feature, see the *IBM Informix ESQL/C Programmer's Manual*. For information about the DataBlade API, a C language API that is provided with IBM Informix, see the *IBM Informix DataBlade API Programmer's Guide*.

Handle non-ASCII characters

The IBM Informix ESQL/C processors obtain the code set for use in Informix ESQL/C source files from the client locale.

Within an Informix ESQL/C source file, you can use non-ASCII characters for the following program objects:

- Informix ESQL/C comments
- Names of SQL identifiers such as databases, tables, columns, views, constraints, prepared statements, and cursors

  For more information, see "Name database objects" on page 3-1.
- Informix ESQL/C host variable and indicator variable names

  For example, in an Informix ESQL/C program, this use of multibyte characters is valid:

  ```
  char A¹A²[20], B¹B²[20];
  EXEC SQL select col1, col2 into :A¹A² :B¹B²;
  ```

  For more information about Informix ESQL/C host variables, see "Non-ASCII characters in host variables" on page 6-2.
- Literal strings

  For example, in an Informix ESQL/C program, the following use of multibyte characters is valid:

  ```
  EXEC SQL insert into tbl1 (nchr1) values 'A¹A²B¹B²';
  ```
- File names and path names, if your operating system supports multibyte characters in file names and path names.

Tip: Some C-language compilers support multibyte characters in literals or comments only. For such compilers, you might need to set the **ESQLMF** and **CC8BITLEVEL** environment variables so that the Informix ESQL/C processor calls a multibyte filter. For more information, see "Generate non-ASCII file names" on page 6-3.

To use non-ASCII characters in your Informix ESQL/C source file, the client locale must support them. For information about the use of non-ASCII characters, see "Non-ASCII characters in identifiers" on page 3-1.

© Copyright IBM Corp. 1996, 2013

Non-ASCII characters in host variables

IBM Informix ESQL/C allows the use of non-ASCII characters in host variables when certain conditions are true.

The following conditions must be true to allow the use of non-ASCII characters:

- The client locale supports a code set with the non-ASCII characters that the host-variable name contains. You must set the client locale correctly before you preprocess and compile an Informix ESQL/C program. For more information, see "Set a GLS locale" on page 1-16.
- Your C compiler supports compilation of the same non-ASCII characters as the source code.

  You must ensure that the C compiler supports use of non-ASCII characters in C source code. For information about how to indicate the support that your C compiler provides for non-ASCII characters, see "Invoke the ESQL/C filter" on page 6-4.

Informix ESQL/C applications can also support non-ASCII characters in comments and SQL identifiers. For more information, see "Non-ASCII characters in identifiers" on page 3-1.

The following code fragment declares an integer host-variable that contains a non-ASCII character in the host-variable name and then selects a serial value into this variable:

```
/*
   This code fragment declares an integer host variable
   "hte_ent", which contains a non-ASCII character in the
   name, and selects a serial value (code number in the
   "numro" column of the "abonns" table) into it.
*/

EXEC SQL BEGIN DECLARE SECTION;
   int hte_ent;
...


EXEC SQL END DECLARE SECTION;
...


EXEC SQL select numro into :hte_ent from abonns
   where nom = 'tker';
```

If the client locale supports the non-ASCII characters, you can use these characters to define indicator variables, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
   char   htevar[30];
   short  ind_de_htevar;
EXEC SQL END DECLARE SECTION;
```

You can then access indicator variables with these non-ASCII names, as the following example shows:

```
:htevar INDICATOR :htevarind

:htevar   :htevar   ind

$htevar      $htevar   ind
```

Generate non-ASCII file names

When an IBM Informix ESQL/C source file is processed, the Informix ESQL/C processor generates a corresponding intermediate file for the source file.

If you use non-ASCII characters (8-bit and multibyte character) in these source file names, the following restrictions affect the ability of the Informix ESQL/C processor to generate file names that contain non-ASCII characters:

- The Informix ESQL/C processor must know whether the operating system is 8-bit clean.

  For more information, see "The GLS8BITFSYS environment variable" on page 2-7.
- The code set of the client locale (the client code set) must support the non-ASCII characters that are used in the Informix ESQL/C source file name.
- Your C compiler supports the non-ASCII characters that the file name of the Informix ESQL/C source file uses.

  If your C compiler does not support non-ASCII characters, you can use the **CC8BITLEVEL** environment variable as a workaround when your source file contains multibyte characters. For more information, see "Generate non-ASCII file names."

Non-ASCII characters in ESQL/C source files

The IBM Informix ESQL/C processor, **esql**, accepts C source programs that are written in the client code set (the code set of the client locale). The Informix ESQL/C preprocessor, **esqlc**, can accept non-ASCII characters (8-bit and multibyte) in the Informix ESQL/C source code as long as the client code set defines them.

The capabilities of your C compiler, however, might limit your ability to use non-ASCII characters within an Informix ESQL/C source file. If the C compiler does not fully support non-ASCII characters, it might not successfully compile an Informix ESQL/C program that contains these characters. To provide support for common non-ASCII limitations of C compilers, Informix ESQL/C provides an Informix ESQL/C filter that is called **esqlmf**.

This section provides the following information about the Informix ESQL/C filter:

- How the Informix ESQL/C filter processes non-ASCII characters
- How you invoke the Informix ESQL/C filter

Filter non-ASCII characters

As part of the compilation process of an IBM Informix ESQL/C source program, the Informix ESQL/C processor calls the C compiler. When you develop Informix ESQL/C source code that contains non-ASCII characters, the way that the C compiler handles such characters can affect the success of the compilation process.

In particular, the following situations might affect compilation of your Informix ESQL/C program:

- Multibyte characters might contain C-language tokens.

  A component of a multibyte character might be indistinguishable from some single-byte characters such as percent (%), comma (,), backslash (\ ), and double quotation mark (") characters. If such characters are included in a quoted string, the C compiler might interpret them as C-language tokens, which can cause compilation errors or even lost characters.
- The C compiler might not be 8-bit clean.

If a code set contains non-ASCII characters (with code values that are greater than 127), the C compiler must be 8-bit clean to interpret the characters. To be 8-bit clean, a compiler must read the eighth bit as part of the code value; it must not ignore or put its own interpretation on the meaning of this eighth bit.

To filter a non-ASCII character, the Informix ESQL/C filter converts each byte of the character to its octal equivalent. For example, suppose the multibyte character $A^1A^2A^3$ has an octal representation of \160\042\244 and appears in the **stcopy()** call.

```
stcopy("A1A2A3", dest);
```

After **esqlmf** filters the Informix ESQL/C source file, the C compiler sees this line as follows:

```
stcopy("\160\042\244", dest); /* correct interpretation */
```

To handle the C-language-token situation, the filter prevents the C compiler from interpreting the A^2 byte (octal \042) as an ASCII double quotation mark and incorrectly parsing the line as follows:

```
stcopy("A1"A3, dest); /* incorrect interpretation of A2 */
```

The C compiler would generate an error for the preceding line because the line has terminated the string argument incorrectly. The **esqlmf** utility also handles the 8-bit-clean situation because it prevents the C compiler from ignoring the eighth bit of the A^3 byte. If the compiler ignores the eighth bit, it incorrectly interprets A^3 (octal \244) as octal \044.

Invoke the ESQL/C filter

The **esql** command can automatically call the IBM Informix ESQL/C filter, **esqlmf**, to process non-ASCII characters.

The following figure shows how an Informix ESQL/C program that contains non-ASCII characters becomes an executable program.

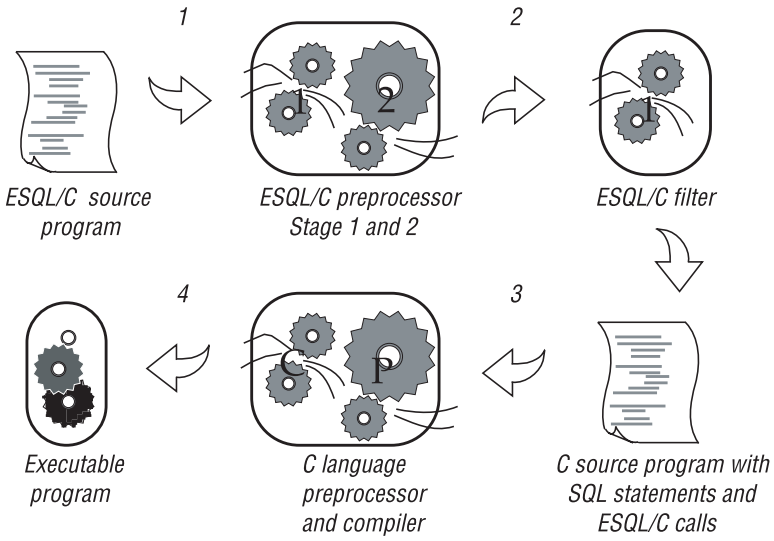


Figure 6-1. Create an ESQL/C executable program from a non-ASCII source program

When you set the following environment variables, you tell **esql** how to invoke **esqlmf**:

- The **ESQLMF** environment variable indicates whether **esql** automatically calls the Informix ESQL/C filter.

  When you set **ESQLMF** to 1, **esql** automatically calls **esqlmf** after the Informix ESQL/C preprocessor and before the C compiler.
- The **CC8BITLEVEL** environment variable indicates the non-ASCII characters in the Informix ESQL/C source file that **esqlmf** filters.

  Set **CC8BITLEVEL** to indicate the ability of your C compiler to process non-ASCII characters.

How **esqlmf** filters an Informix ESQL/C source file depends on the value of the **CC8BITLEVEL** environment variable. For each value of **CC8BITLEVEL**, the following table shows the **esqlmf** command that the Informix ESQL/C processor invokes on an Informix ESQL/C source file.

CC8BITLEVEL	The esqlmf action
0	Converts all non-ASCII characters, in literal strings and comments, to octal constants.
1	Converts non-ASCII characters in literal strings, but not in comments, to octal constants.
2	Converts non-ASCII characters in literal strings to octal constants to ensure that all the bytes in the non-ASCII characters have the eighth bit set.
3	Does not invoke **esqlmf**.

Important: To start the **esqlmf** commands that **CC8BITLEVEL** can specify, you must set the **ESQLMF** environment variable to 1.

When you set **CC8BITLEVEL** to 0, 1, or 2, the Informix ESQL/C processor performs the following steps:

1. Converts the embedded-language statements (source.ec) to C-language source code (source.c) with the Informix ESQL/C preprocessor
2. Filters non-ASCII characters in the preprocessed file (source.c) with the Informix ESQL/C filter, **esqlmf** (if the **ESQLMF** environment variable is 1)

   Before **esqlmf** begins filtering, it creates a copy of the C source file (source.c) that has the .c_ file extension (source.c_).
3. Compiles the filtered C source file (source.c) with the C compiler to create an object file (source.o)
4. Links the object file with the Informix ESQL/C libraries and your own libraries to create an executable program

When you set **CC8BITLEVEL** to 3, the Informix ESQL/C processor omits step 2 in the preceding list.

If you do not set **CC8BITLEVEL**, then **esql** converts non-ASCII characters in literal strings and comments. You can modify the value of **CC8BITLEVEL** to reflect the capabilities of your C compiler.

Define variables for locale-sensitive data

The SQL data types NCHAR and NVARCHAR support locale-specific data, in the sense that the database server uses localized collation (if the locale defines localized collation), rather than code set order, for sorting data strings of these types.

For more information about NCHAR and NVARCHAR data types, see "Character data types" on page 3-6.

IBM Informix ESQL/C supports the predefined data types **string**, **fixchar**, and **varchar** for host variables that contain character data. In addition, you can use the C **char** data type for host variables. You can use these four host-variable data types for NCHAR and NVARCHAR data.

Your Informix ESQL/C program can access columns of data types NCHAR and NVARCHAR when it selects into or reads from character host variables. The following code fragment declares a **char** host variable, **hte**, and then selects NCHAR data into the **hte** variable:

```
/*
   This code fragment declares a char host variable "hte",
   which contains a non-ASCII character in the name, and
   selects NCHAR data (non-ASCII names in the "nom" column
   of the "abonns" table) into it.
*/

EXEC SQL BEGIN DECLARE SECTION;
   char hte[10];
...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL select nom into :hte from abonns
   where numro > 13601;
```

When you declare Informix ESQL/C host variables for the NCHAR and NVARCHAR data types, note the relationship between the declared size of the variable and the amount of character data that it can hold, as follows:

- If your locale supports a single-byte code set, the size of the NCHAR and NVARCHAR variable determines the number of characters that it can hold.
- If your locale supports a multibyte code set, you can no longer assume a one-byte-per-character relationship.

  In this case, you must ensure that you declare an Informix ESQL/C host variable large enough to accommodate the number of characters that you expect to receive from the database.

For more information, see "The NCHAR data type" on page 3-6 and "The NVARCHAR data type" on page 3-8.

You can insert a value that a character host variable (**char**, **fixchar**, **string**, or **varchar**) holds in columns of the NCHAR or NVARCHAR data types.

Enhanced ESQL/C library functions

IBM Informix SQL API products support locale-specific enhancements to the Informix ESQL/C library functions.

These Informix ESQL/C library functions fall into the following categories:

- DATE-format functions
- DATETIME-format functions
- Numeric-format functions
- String functions

In addition, this section describes the GLS-related error messages that these Informix ESQL/C functions might produce.

DATE-format functions

There are several IBM Informix ESQL/C DATE-format functions that support extensions to format era-based DATE values.

The Informix ESQL/C DATE-format functions are as follows:

- **rdatestr()**
- **rstrdate()**
- **rdefmtdate()**
- **rfmtdate()**

These functions support extensions to format era-based DATE values:

- Support for the **GL_DATE** environment variable
- Era-based date formats of the **DBDATE** environment variable
- Extensions to the date-format strings for Informix ESQL/C DATE-format functions
- Support for a precedence of date end-user formats

These topics describe locale-specific behavior of the Informix ESQL/C DATE-format functions. For details, see the *IBM Informix ESQL/C Programmer's Manual*.

GL_DATE support

The **GL_DATE** setting can affect the results that the IBM Informix ESQL/C DATE-format functions generate.

The end-user format that **GL_DATE** specifies overrides date end-user formats that the client locale defines. For more information, see "Precedence for date end-user formats" on page 6-9.

DBDATE extensions

When you set **DBDATE** to one of the era-based formats, the functions use era-based dates to convert between date strings and internal DATE values.

The IBM Informix ESQL/C DATE-format functions that support the extended era-based date syntax for the **DBDATE** environment variable are as follows:

- **rdatestr()**
- **rstrdate()**

The following Informix ESQL/C example shows a call to the **rdatestr()** library function:

```
char str[100];
long jdate;
...
rdatestr(jdate, str);
printf("%s\n", str);
```

If you set **DBDATE** to GY2MD/ and **CLIENT_LOCALE** to the Japanese SJIS locale (**ja_jp.sjis**), the preceding code prints this value for the date 08/18/1990:

```
H02/08/18
```

Important: IBM Informix products treat any undefined characters in the alphabetic era specification as an error.

If you set **DBDATE** to an era-based date format (which is specific to a Chinese or Japanese locale), make sure to set the **CLIENT_LOCALE** environment variable to a locale that supports era-based dates.

Extended DATE-format strings

The IBM Informix ESQL/C DATE-format functions that support the extended-DATE format strings are **rdefmtdate()** and **rfmtdate()**.

The following table shows the extended-format strings that these Informix ESQL/C functions support for use with GLS locales. These extended-format strings format eras with two-digit year offsets.

Era year	Format	Era used
Full era year: full name of the base year (period) followed by a two-digit year offset. Same as **GL_DATE** end-user format of "%EC%02.2Ey"	eyy	The era that the client locale (which **CLIENT_LOCALE** indicates) defines
Abbreviated era year: abbreviated name of the base year (period) followed by a two-digit year offset. Same as **GL_DATE** end-user format of "%Eg%02.2Ey"	gyy	The era that the client locale (which **CLIENT_LOCALE** indicates) defines

The following table shows some extended-format strings for era-based dates. These examples assume that the client locale is Japanese SJIS (**ja_jp.sjis**).

Description	Example format	October 5, 1990 prints as:
Abbreviated era year	gyymmd	H021005
	gyy.mm.dd	H02.10.05
Full era year	eyymmdd	A1A2021005
	eyy-mm-dd	A1A202-10-05
	eyyB^{1}B^{2}mmB^{1}B^{2}ddB^{1}B^2	A1A202B1B210B1B205B1B2

The following Informix ESQL/C code fragment contains a call to the **rdefmtdate()** library function:

```
char fmt_str[100];
char in_str[100];
long jdate;
...
```

```
rdatestr("10/05/95", &jdate);
stcopy("gyy/mm/dd", fmt_str);
rdefmtdate(&jdate, fmt_str, in_str);
printf("Abbreviated Era Year: %s\n", in_str);

stcopy("eyymmdd", fmt_str);
rdefmtdate(&jdate, fmt_str, in_str);
printf("Full Era Year: %s\n", in_str);
```

When the **CLIENT_LOCALE** specifies the Japanese SJIS (**ja_jp.sjis**) locale, the code fragment displays the following output:

```
Abbreviated Era Year: H07/10/05
Full Era Year: H021005
```

Precedence for date end-user formats

The IBM Informix ESQL/C DATE-format functions use a precedence to determine the end-user format for values in DATE columns:

The precedence is as follows:

1. The end-user format that **DBDATE** specifies (if **DBDATE** is set)
2. The end-user format that **GL_DATE** specifies (if **GL_DATE** is set)
3. The end-user date format that the client locale specifies (if **CLIENT_LOCALE** is set)
4. The end-user date format from the default locale: %m %d %iY

For more information about the precedence of **DBDATE**,**GL_DATE** and **CLIENT_LOCALE**, see "Date and time precedence" on page 1-33.

Tip: IBM Informix products support **DBDATE** for compatibility with earlier products. It is recommended that you use the **GL_DATE** environment variable for new client applications.

DATETIME-format functions

The IBM Informix ESQL/C DATETIME-format functions are **dtcvfmtasc( )** and **dttofmtasc( )**.

These functions support extensions to format era-based DATETIME values:

- Support for the **GL_DATETIME** environment variable
- Support for era-based date and times of the **DBTIME** environment variable
- Extensions to the date and time format strings for Informix ESQL/C DATETIME-format functions
- Support for a precedence of DATETIME end-user formats

These topics describe locale-specific behavior of the Informix ESQL/C DATETIME-format functions. For general information about the Informix ESQL/C DATETIME-format functions, see the *IBM Informix ESQL/C Programmer's Manual*.

GL_DATETIME support

The **GL_DATETIME** setting can affect results that the Informix ESQL/C DATETIME-format functions generate.

The end-user format that **GL_DATETIME** specifies overrides date and time formats that the client locale defines. For more information, see "Precedence for DATETIME end-user formats" on page 6-10.

DBTIME support

The IBM Informix ESQL/C DATETIME-format functions support the extended era-based date and time format strings for the **DBTIME** environment variable.

When you set **DBTIME** to an era-based format, these functions can convert between literal DATETIME strings and internal DATETIME values.

Tip: IBM Informix products support **DBTIME** for compatibility with earlier products. It is recommended that you use the **GL_DATETIME** environment variable for new applications.

If you set **DBTIME** to an era-based DATETIME format (which is specific to a Chinese or Japanese locale), make sure to set the **CLIENT_LOCALE** environment variable to a locale that supports era-based dates and times.

Extended DATETIME-format strings

IBM Informix ESQL/C DATETIME-format functions support extended-format strings.

The following table shows the extended-format strings that the Informix ESQL/C DATETIME-format functions support.

Format	Description	December 27, 1991 printed
%y %m %dc1	Taiwanese Ming Guo date	80 12 27
%Y %m %dc1	Taiwanese Ming Guo date	0080 12 27
%y %m %dj1	Japanese era with abbreviated era symbols	H03 12 27
%Y %m %dj1	Japanese era with abbreviated era symbols	H0003 12 27
%y %m %dj2	Japanese era with full era symbols	A1A2B1B203 12 27
%Y %m %dj2	Japanese era with full era symbols	A1A2B1B20003 12 27

In addition to the formats in the preceding table, these Informix ESQL/C DATETIME-format functions support the GLS date and time specifiers. For a list of these specifiers, see "The GL_DATE environment variable" on page 2-10 and "The GL_DATETIME environment variable" on page 2-15.

Precedence for DATETIME end-user formats

The IBM Informix ESQL/C DATETIME-format functions use a precedence to determine the end-user format of values in DATETIME columns.

The precedence is as follows:

1. The end-user format that **DBTIME** specifies (if **DBTIME** is set)
2. The end-user format that **GL_DATETIME** specifies (if **GL_DATETIME** is set)
3. The date and time end-user formats that the client locale specifies (if **CLIENT_LOCALE** is set)
4. The date and time end-user format from the default locale: %iY-%m-%d %H:%M:%S

For more information about the precedence of **DBDATE**, **GL_DATE**, and **CLIENT_LOCALE**, see "Date and time precedence" on page 1-33.

Numeric-format functions

The IBM Informix ESQL/C numeric-format functions are **rfmtdec()**, **rfmtdouble()**, and **rfmtlong()**.

These functions support the following extensions to format numeric values:

- Support for multibyte characters in format strings
- Locale-specific formats for numeric values
- Formatting characters for currency symbols
- Support for the **DBMONEY** environment variable

These topics describe locale-specific behavior of the Informix ESQL/C numeric-format functions. For general information about the Informix ESQL/C numeric-format functions, see the *IBM Informix ESQL/C Programmer's Manual*.

Tip: For a list of errors that these Informix ESQL/C numeric-format functions might return, see "GLS-specific error messages" on page 6-14.

Support for multibyte characters

The IBM Informix ESQL/C numeric-format functions support multibyte characters in their format strings if your client locale supports a multibyte code set that defines these characters.

These Informix ESQL/C functions and routines, however, interpret multibyte characters as literal characters. You cannot use multibyte equivalents of the ASCII formatting characters.

For example, the following Informix ESQL/C code fragment shows a call to the **rfmtlong( )** function with the multibyte character A^1A^2 in the format string:

```
stcopy("A1A2***,***", fmtbuf);
rfmtlong(78941, fmtbuf, outbuf);
printf("Formatted value: %s\n", outbuf);
```

This code fragment generates the following output (if the client code set contains the A^1A^2 character):

```
Formatting value: A1A2*78,941
```

Locale-specific numeric formatting

The IBM Informix ESQL/C numeric-format functions require a format string as an argument.

This format string determines how the numeric-format function formats the numeric value. A format string consists of a series of formatting characters and the following currency notation.

Formatting character	Function
Dollar sign ($)	Currency symbol
Comma (,)	Thousands separator
Period (.)	Decimal separator

Regardless of the client locale that you use, you must use the preceding ASCII symbols in the format string to identify where to place the currency symbol, decimal separator, and thousands separator. The numeric-format function uses the following precedence to translate these symbols to their locale-specific equivalents:

1. The symbols that **DBMONEY** indicates (if **DBMONEY** is set)

   For information about the locale-specific behavior of **DBMONEY**, see "DBMONEY extensions" on page 6-13.
2. The symbols that the appropriate locale category of the client locale (if **CLIENT_LOCALE** is set) specifies

   If the format string contains either a $ or @ formatting character, a numeric-format function assumes that the value is a monetary value and refers to the MONETARY category of the client locale. If these two symbols are not in the format string, a numeric-format function refers to the NUMERIC category of the client locale.

   For more information about the use of the $ and @ formatting characters, see "Currency-symbol formatting." For more information about the MONETARY and NUMERIC locale categories, see "Locale categories" on page A-2.
3. The actual symbol that appears in the format string ($, comma, or period)

These numeric-format functions replace the dollar sign in the format string with the currency symbol that **DBMONEY** specifies (if it is set) or with the currency symbol that the client locale specifies (if **DBMONEY** is not set).

The same is true for the decimal separator and thousands separator. For example, the following Informix ESQL/C code fragment shows a call to the **rfmtlong()** function:

```
stcopy("$***,***.&&", fmtbuf);
rfmtlong(78941, fmtbuf, outbuf);
printf("Formatted value: %s\n", outbuf);
```

In the default, German, and Spanish locales, this code fragment produces the following results for the logical MONEY value of `78941.00` (if **DBMONEY** is not set).

Format string	Client locale	Formatted Value
$***,***.&&	Default locale (**en_us.8859-1**)	$*78,941.00
	German locale (**de_de.8859-1**)	DM*78.941,00
	Spanish locale (**es_es.8859-1**)	Pts*78.941,00

Currency-symbol formatting

The IBM Informix ESQL/C numeric-format functions support all formatting characters that the *IBM Informix ESQL/C Programmer's Manual* describes.

In addition, you can use the following formatting characters to indicate the placement of a currency symbol in the formatted output.

Formatting character	Function
$	This character is replaced by the *front* currency symbol if the locale defines one. The MONETARY category of the locale defines the *front* currency symbol, which is the symbol that appears before a monetary value. When you group several dollar signs in a row, a single currency symbol floats to the rightmost position that it can occupy without interfering with the number.
@	This character is replaced by the *back* currency symbol if the locale defines one. The MONETARY category of the locale defines the *back* currency symbol, the symbol that appears after a monetary value.

For more information, see "The MONETARY category" on page A-5.

You can include both formatting characters in a format string. The locale defines whether the currency symbol appears before or after the monetary value, as follows:

- If the locale formats monetary values with a currency symbol before the value, the locale sets the currency symbol to the *front* currency symbol and sets the *back* currency symbol to a blank character.
- If the locale formats monetary values with a currency symbol after the value, the locale sets the currency symbol to the *back* currency symbol and sets the *front* currency symbol to a blank character.

The default locale defines the currency symbol as the *front* currency symbol, which appears as a dollar sign ($). In the default locale, the *back* currency symbol appears as a blank space. In the default, British, and French locales, the numeric-format functions produce the following results for the internal MONEY value of `1.00`.

Format string	Client locale	Formatted result
$***,***	Default locale (**en_us.8859-1**) British locale (**en_gb.8859-1**) French locale (**fr_fr.8859-15**)	$******1 £******1 s******1
$***,***@	Default locale (**en_us.8859-1**) British locale (**en_gb.8859-1**) French locale (**fr_fr.8859-15**)	$******1s £******1s s******1€
$$,$$$.$$	Default locale (**en_us.8859-1**) British locale (**en_gb.8859-1**) French locale (**fr_fr.8859-15**)	ssss$1.00 ssss£1.00 sssss1,00
,@	Default locale (**en_us.8859-1**) British locale (**en_gb.8859-1**) French locale (**fr_fr.8859-15**)	******1s ******1s ******1€
@***,***	Default locale (**en_us.8859-1**) British locale (**en_gb.8859-1**) French locale (**fr_fr.8859-15**)	s******1 s******1 €******1

In the preceding table, the character `s` represents a blank or space, € is the currency symbol for Euros, and £ is the British currency symbol for pounds Sterling.

The **DBMONEY** environment variable can also set the precede-currency symbol and the succeed-currency symbol. The syntax diagram in "The DBMONEY environment variable" on page 2-5 refers to these symbols as *front* and *back*. The **DBMONEY** setting, if one is specified, takes precedence over the symbols that the MONETARY category of the locale defines.

DBMONEY extensions

You can specify the currency symbol and decimal-separator symbol with the **DBMONEY** environment variable. These settings override any currency notation that the client locale specifies.

You can use multibyte characters for these symbols if your client code set supports them. For example, the following table shows how multibyte characters appear in examples of output.

Format string	Number to format	DBMONEY	Output
"$$,$$$.$$"	1234	'$'.	$1,234.00
"$$,$$$.$$"	1234	DM,	DM1.234,00
"$$,$$$.$$"	1234	A^1A^2.	A1A21,234.00
"$$,$$$.$$"	1234	.A^1A^2	s1,234.00
"&&,&&&.&&@"	1234	.A^1A^2	s1,234.00A1A2
"$&&,&&&.&&@"	1234	A^1A^2.	A1A2s1,234.00
"$&&,&&&.&&@"	1234	.A^1A^2	s1,234.00A1A2
"@&&,&&&.&&"	1234	.A^1A^2	A1A2s1,234.00

In the preceding table, the character s represents a blank or space.

String functions

The **rdownshift( )** and **rupshift ( )** IBM Informix ESQL/C string functions support locale-specific shifted characters.

These string functions use the information in the CTYPE category of the client locale to determine the shifted code points. If the client locale specifies a multibyte code set, these functions can operate on multibyte strings.

Important: With multibyte character strings, a shifted string might occupy more memory after a shift operation than before. You must ensure that the buffer you pass to these Informix ESQL/C shift functions is large enough to accommodate this expansion.

GLS-specific error messages

The IBM Informix ESQL/C DATE-format, DATETIME-format, and Numeric-format functions might generate GLS-specific error messages.

For more information about GLS-specific error messages, use the **finderr** utility on UNIX or the **Informix Error Messages** utility on Windows.

Handle code-set conversion

When the client and database code sets differ, the IBM Informix ESQL/C client application performs code-set conversion on character data.

For more information, see "Perform code-set conversion" on page 1-29.

If your Informix ESQL/C application executes in an environment in which code-set conversion might occur, check that the application correctly handles the following situations:

- When the application writes simple large objects (TEXT or BYTE data) to the database, it must set the **loc_type** field in the locator structure **loc_t** to indicate the type of simple large object that it needs to write.
- When the application writes smart large objects (CLOB or BLOB data) to the database, it uses various large-object file descriptors.
- When the application uses the **sqlda** structure to describe dynamic SQL statements, it must account for possible size differences in character data.

- When the application has character data that might undergo code-set conversion, you must declare character buffers that can hold the data.

  For more information, see "Avoid partial characters" on page 5-7.

Writing TEXT values

IBM Informix ESQL/C uses the **loc_t** locator structure to read simple large objects from and write simple large objects to the database server.

The **loc_type** field of this structure indicates the data type of the simple large object that the structure describes. When the client and database code sets are the same (no code-set conversion), the client application does not need to set the **loc_type** field explicitly because the database server can determine the simple large object data type implicitly. The database server assumes character data in the TEXT data type and noncharacter data in the BYTE data type.

If the client and database code sets are different and convertible, however, the client application must know the data type of the simple large object in order to determine whether to perform code-set conversion on the data.

Before an Informix ESQL/C client application inserts a simple large object in the database, it must explicitly set the **loc_type** field of the simple large object:

- For a TEXT value, the Informix ESQL/C client application must set the **loc_type** field to SQLTEXT before the INSERT statement. The client performs code-set conversion on TEXT data before it sends this data to the database for insertion.
- For a BYTE value, the Informix ESQL/C client application must set the **loc_type** field to SQLBYTES before the INSERT statement. The client does not perform code-set conversion on BYTE data before it sends this data to the database for insertion.

Important: The `sqltypes.h` header file defines the data type constants SQLTEXT and SQLBYTES. To use these constants, you must include this header file in your Informix ESQL/C source file.

Your Informix ESQL/C source code does not need to set **loc_type** before it reads simple-large-object data from a database. The database server obtains the data type of the simple large object from the database and sends this data type to the client with the data.

If you set **loc_bufsize** to `-1`, Informix ESQL/C allocates memory to hold a single simple large object. It stores the address of this memory buffer in the **loc_buffer** field of the **loc_t** structure. If the client application performs code-set conversion on TEXT data that the database server retrieves, Informix ESQL/C handles any possible data expansion as follows:

1. Frees the existing memory that the **loc_buffer** field references
2. Reallocates a memory buffer that is large enough to store the expanded TEXT data
3. Assigns the address of this new buffer to the **loc_buffer** field
4. Assigns the size of the new memory buffer to the **loc_bufsize** field

If this reallocation occurs, Informix ESQL/C changes the memory address at which it stores the TEXT data. If your Informix ESQL/C program references this address, the program must account for the address change.

Informix ESQL/C does not need to reallocate memory for the TEXT data if code-set conversion does not expand the TEXT data or if it condenses the data. In either of these cases, the **loc_buffer** field remains unchanged, and the **loc_bufsize** field contains the size of the buffer that the **loc_buffer** field references.

The DESCRIBE statement

The **sqlda** structure is a dynamic-management structure that contains information about columns in dynamic SQL statements. The DESCRIBE...INTO statement uses the **sqlda** structure to return information about the columns in the select list of the Projection clause of a SELECT statement.

It sets the **sqlvar** field of an **sqlda** structure to point to a sequence of partially filled **sqlvar_struct** structures. Each structure describes a single select-list column.

Each **sqlvar_struct** structure contains character data for the column name and the column data. When the IBM Informix ESQL/C client application fills this structure, the column name and the column data are in the client code set. When the database server fills this structure and executes a DESCRIBE...INTO statement, this character data is in the database code set.

When the client application performs code-set conversion between the client and database code sets, the number of bytes that is required to store the column name and column data in the client code set might not equal the number that is required to store this same information in the database code set. Therefore, the size of the character data in **sqlvar_struct** might increase or decrease during code-set conversion. To handle this possible difference in size, the client application must ensure that it correctly handles the character data in the **sqlvar_struct** structure.

The sqldata field

To hold the column data, the client application must allocate a buffer and set **sqldata** to point to this buffer. If your client application might perform code-set conversion, it must allocate sufficient storage to handle the increase in the size of the column data that might occur.

When the DESCRIBE ... INTO statement sets the **sqllen** field, the **sqllen** value indicates the length of the column data in the database code set. Therefore, if you use the value of **sqllen** that the DESCRIBE ... INTO statement retrieves, you might not allocate a buffer that is sufficiently large for the data values when they are in the client code set.

For example, the following code fragment allocates an **sqldata** buffer with the **malloc( )** system call:

```
EXEC SQL include sqlda;
...
struct sqlda *q_desc;
...
EXEC SQL describe sqlstmt_id into q_desc;
...
q_desc->sqlvar[0].sqldata =
   (char *)malloc(q_desc->sqlvar[0].sqllen);
```

In the preceding code fragment, the client application might truncate characters that it converts because the client application uses the **sqllen** value to determine the buffer size. Instead, increase the buffer to four times its original size when you allocate a buffer, as the following code fragment shows:

```
EXEC SQL include sqlda;
EXEC SQL define BUFSIZE_FACT 4;
...

struct sqlda *q_desc;
...

q_desc->sqlvar[0].sqllen =
   q_desc->sqlvar[0].sqllen * BUFSIZE_FACT + 1;
q_desc->sqlvar[0].sqldata =
   (char *)malloc(q_desc->sqlvar[0].sqllen);
```

A buffer-size factor (BUFSIZE_FACT) of 4 is suggested because a multibyte character has a maximum size of 4 bytes.

The sqlname field

The **sqlname** field contains the name of the column.

When the client application performs code-set conversion, this column name might also undergo expansion when the application converts it from the database code set to the client code set. Because the Informix ESQL/C application stores the buffer for **sqlname** data in its internal work area, your Informix ESQL/C source code does not have to handle possible buffer-size increases. Your code processes the contents of **sqlname** in the client code set.

The TRIM function

When you dynamically execute a SELECT statement, the DESCRIBE statement can return information about the columns in the select list of the Projection clause at run time. DESCRIBE returns the data type of a select-list column in the appropriate field of the dynamic-management structure that you use.

When you use the DESCRIBE statement on a prepared SELECT statement with the TRIM function in its select list, the data type of the trimmed column that DESCRIBE returns depends on the database server that you use and the data type of the column to be trimmed (the *source character-value expression*). For more information about the source character-value expression, see the description of the TRIM function in the *IBM Informix Guide to SQL: Syntax*.

The data type that the DESCRIBE statement returns depends on the data type of the source character-value expression, as identified in the following table:

Table 6-1. The TRIM function

Sequence number	Operand type	Result type	Result length
1	(N)CHAR(1-255)	(N)VARCHAR	Up to 255
2	(N)CHAR(>255)	LVARCHAR	Up to 32739
3	(N)VARCHAR	(N)VARCHAR	Up to 255
4	LVARCHAR	LVARCHAR	Up to 32739

The following SELECT statement contains the **manu_code** column, which is defined as a CHAR data type, and the **cat_advert** column, which is defined as a VARCHAR column. When you describe the following SELECT statement and use the TRIM function, DESCRIBE returns a data type of SQLVCHAR for both trimmed columns:

```
SELECT TRIM(manu_code), TRIM(cat_advert) FROM catalog;
```

If the **manu_code** column is defined as NCHAR instead, DESCRIBE returns a data type of SQLNVCHAR for this trimmed column.

Important: The `sqltypes.h` header file defines the data type constants SQLCHAR, SQLVCHAR, and SQLNVCHAR. To use these constants, include this header file in your Informix ESQL/C source file.

Appendix A. Manage GLS files

These topics describe the files provided for GLS, which are executable only.

Access GLS files

IBM Informix products access the GLS files to obtain locale-related information.

The following table shows the GLS files to obtain locale-related information. For an overview of what type of information these files provide, see "A GLS locale" on page 1-9.

GLS files	Reference
GLS locale files	"GLS locale files" on page A-2
Code-set-conversion files	"Code-set-conversion files" on page A-8
Code-set files	"Code-set files" on page A-10
The `registry` file	"The IBM Informix registry file (Windows)" on page A-10

In general, you do not need to examine the GLS files. You might, however, want to look at these files to determine the following locale-specific information.

Locale-specific information	GLS file to examine	Reference
Exact localized collation order	Source locale file (`*.lc`): COLLATION category	"The COLLATION category" on page A-3
Exact code-set collation order	Source code-set file (`*.cm`)	"Code-set files" on page A-10
Locale-specific mapping between uppercase and lowercase characters	Source locale file (`*.lc`): CTYPE category	"The CTYPE category" on page A-3
Locale-specific classification of characters	Source locale file (`*.lc`): CTYPE category	"The CTYPE category" on page A-3
Code-set-specific character mappings	Source code-set file (`*.cm`)	"Code-set files" on page A-10
Mappings between characters of the source and target code sets	Source code-set-conversion file (`*.cv`)	"Code-set-conversion files" on page A-8
Method for character mismatches during code-set conversion	Source code-set-conversion file (`*.cv`)	"Code-set-conversion files" on page A-8
Code points for characters	Source code-set file (`*.cm`)	"Code-set files" on page A-10

© Copyright IBM Corp. 1996, 2013

Locale-specific information	GLS file to examine	Reference
Numeric (nonmonetary) data end-user format	Source locale file (*.lc): NUMERIC category	"The NUMERIC category" on page A-4
Monetary data end-user format	Source locale file (*.lc): MONETARY category	"The MONETARY category" on page A-5
Date data end-user format	Source locale file (*.lc): TIME category	"The TIME category" on page A-5
Time data end-user format	Source locale file (*.lc): TIME category	"The TIME category" on page A-5

GLS locale files

The *locale file* defines a GLS locale. It describes the basic language and cultural conventions that are relevant to the processing of data for a given language and territory.

These topics describe the locale categories and the locations of the locale files.

Locale categories

A locale file specifies behaviors for the locale categories.

The CTYPE and COLLATION categories primarily affect how the database server stores and retrieves character data in a database. The NUMERIC, MONETARY, and TIME categories affect how a client application formats the internal values of the associated SQL data types. For more information about end-user formats, see "End-user formats" on page 1-13 and "Customize end-user formats" on page 1-32. The following table describes the locale categories and the behaviors for the default locale, U.S. English.

Locale category	Description	In default locale (U.S. English)
CTYPE	Controls the behavior of character classification and case conversion.	The default code set classifies characters. On UNIX, the default code set is ISO8859-1. On Windows, the default code set is Windows Code Page 1252.
COLLATION	Controls the behavior of string comparisons.	The default locale does not define a localized order. Therefore, the database server collates NCHAR and NVARCHAR data in code-set order (unless SET COLLATION has specified some localized order).

Locale category	Description	In default locale (U.S. English)
NUMERIC	Controls the behavior of non-monetary numeric end-user formats.	The following numeric notation for use in numeric end-user formats: • Thousands separator: comma (,) • Decimal separator: period (.) • Number of digits between thousands separators: 3 • Symbol for positive number: plus (+) • Symbol for negative number: minus (-) • No alternative digits for era-based dates
MONETARY	Controls the behavior of currency end-user formats.	The following currency notation for use in monetary end-user formats: • Currency symbol: dollar sign ($) appears as the *front* symbol before the currency value • No *back* currency symbol is defined. • Thousands separator: comma (,) • Decimal separator: period (.) • Number of digits between thousands separators: 3 • Symbol for positive number: plus (+) • Symbol for negative number: minus (-) Default scale for MONEY columns: 2
TIME	Controls the behavior of date and time end-user formats.	The following date and time end-user formats: • DATE values: %m/%d/%iy • DATETIME values: %iY-%m-%d %H:%M:%S No definitions for era-based dates.
MESSAGES	Controls the definitions of affirmative and negative responses to messages.	None

The CTYPE category

The CTYPE category defines how to classify the characters of the code set that the locale supports.

This category includes specifications for which characters the locale classifies as spaces, blanks, control characters, digits, uppercase letters, lowercase letters, and punctuation symbols.

This category might also include mappings between uppercase and lowercase letters. IBM Informix products access this category when they need to determine the validity of an identifier name, to shift the letter case of a character, or to compare characters.

The COLLATION category

The COLLATION category can define a localized order.

When an IBM Informix product needs to compare two strings, it first breaks up the strings into a series of collation elements. The database server compares each pair of collation elements according to the collation weights of each element. The COLLATION category supports the following capabilities:

- Multicharacter collation elements define sets of characters that the database server should collate as a single unit. For example, the localized collating order might treat the Spanish double-L (ll) as a single collation element instead of as a pair of l's.
- Equivalence classes assign the same collation weight to different elements. For example, the localized order might specify that a and A are an equivalence class (a and A are equivalent characters).

The difference in collation order is the only distinction between the CHAR and NCHAR data types and the VARCHAR and NVARCHAR data types. For more information, see "Character data types" on page 3-6.

If a locale does not contain a COLLATION category, IBM Informix products use code-set order for collation of all character data types:

- CHAR
- LVARCHAR
- NCHAR
- NVARCHAR
- TEXT
- VARCHAR

The SET COLLATION statement can specify a localized collation that is different from the COLLATION setting of the locale that **DB_LOCALE** specifies. The scope of the collating order that SET COLLATION specifies is the current session, but database objects that can sort strings, such as constraints, indexes, UDRs, and triggers, always use the collating order from the time of their creation when they sort NCHAR or NVARCHAR values.

The NUMERIC category

The NUMERIC category defines the numeric notation for end-user formats of nonmonetary numeric values.

The numeric notation for end-user formats of nonmonetary numeric values are:

- The numeric decimal separator
- The numeric thousands separator
- The number of digits to group together before inserting a thousands separator
- The characters that indicate positive and negative numbers

This numeric notation applies to the end-user formats of data for numeric (DECIMAL, INTEGER, SMALLINT, FLOAT, SMALLFLOAT) columns within a client application.

Important: Information in the NUMERIC category does not affect the internal format of the numeric data types in the database.

The NUMERIC category also defines alternative digits for use in era-based dates and times. For information about alternative digits, see "Alternative date formats" on page 2-12 and "Alternative time formats" on page 2-17.

The MONETARY category

The MONETARY category defines the currency notation for end-user formats of monetary values.

The currency notion for end-user formats of monetary values are:

- The currency symbol, and whether it appears before or after a monetary value
- The monetary decimal separator
- The monetary thousands separator
- The number of digits to group between each appearance of a monetary thousands separator
- The characters that indicate positive and negative monetary values and the position of these characters (before or after)
- The scale (the number of fractional digits to the right of the decimal point) to display

This currency notation applies to the end-user formats of data from MONEY columns within a client application.

Important: Information in the MONETARY category does not affect the internal format of the MONEY data type in the database.

The MONETARY category also defines the default scale for a MONEY column. For the default locale (U.S. English), the database server stores values of the data type MONEY(*precision*) in the same internal format as the data type DECIMAL(*precision*,2). A nondefault locale can define a different default scale. For more information about default scales, see "Specify values for the scale parameter" on page 3-31.

The TIME category

The TIME category lists characters and symbols that format date and time values.

This information includes the names and abbreviations for days of the week and months of the year. It also includes special representations for dates, time (12-hour and 24-hour), and DATETIME values.

These representations can include the names of eras (as in the Japanese Imperial era system) and non-Gregorian calendars (such as the Arabic lunar calendar). The locale specifies the calendar (Gregorian, Hebrew, Arabic, Japanese Imperial, and so on) for reading or printing a month, day, or year.

If the locale supports era-based dates and times, the TIME category defines the full and abbreviated era names and special date and time representations. For more information, see "Alternative date formats" on page 2-12 and "Alternative time formats" on page 2-17.

This date and time information applies to the end-user formats of data in DATE and DATETIME columns within a client application.

Important: Information in the TIME category does not affect the internal format of the DATE and DATETIME data types in the database.

The MESSAGES category

The MESSAGES category defines the format for affirmative and negative responses.

This category is optional. IBM Informix products do not use the strings that the MESSAGES category defines.

To obtain the locale name for the MESSAGES category of the client locale, a client application uses the locale that **CLIENT_LOCALE** indicates. If **CLIENT_LOCALE** is not set, the client sets the category to the default locale.

Location of locale files

When an IBM Informix product needs to obtain locale-specific information, it accesses one of the GLS locale files.

IBM Informix access one of the files in the following table.

Platform	Locale file
UNIX	`$INFORMIXDIR/gls/lcX/lg_tr/codemodf.lco`
Windows	`%INFORMIXDIR%\gls\lcX\lg_tr\codemodf.lco`

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM Informix product, and `gls` is the subdirectory that contains the GLS files. This rest of this section describes the remaining elements in the path name of GLS locale files.

Locale-file subdirectories

The subdirectories of the `lcX` subdirectory, where `X` represents the version number for the locale object-file format, contain the GLS locale files. These subdirectories have names of the form `lg_tr`, where `lg` is the 2-character language name and `tr` is the 2-character territory name that the locale supports.

The next table shows some languages and territories that IBM Informix products can support, and their associated locale-file subdirectory names.

Language	Territory	Locale-file subdirectory
English	Australia	en_au
	United States	en_us
	Great Britain	en_gb
German	Germany	de_de
	Austria	de_at
	Switzerland	de_ch
French	Belgium	fr_be
	Canada	fr_ca
	Switzerland	fr_ch
	France	fr_fr

Locale source and object files

Each locale file has two forms.

Each locale file has the following two forms:

- A locale *source* file is an ASCII file that defines the locale categories for the locale.

  This file has the `.lc` file extension and serves as documentation for the corresponding object file.
- A locale *object* file is a compiled form of the locale information.

  IBM Informix products use the object file to obtain locale information quickly. Locale object files have the `.lco` file extension.

The header of the locale source file (`.lc`) lists the language, territory, code set, and any optional locale modifier of the locale. A section of the locale source file supports each of the locale categories, unless that category is empty, as the next table shows.

Locale category	Reference		Locale category	Reference
CTYPE	"The CTYPE category" on page A-3		MONETARY	"The MONETARY category" on page A-5
COLLATION	"The COLLATION category" on page A-3		TIME	"The TIME category" on page A-5
NUMERIC	"The NUMERIC category" on page A-4		MESSAGES	"The MESSAGES category" on page A-5

Locale file names

To conform to the `8.3 filename.ext` restriction on the maximum number of characters in valid file names and file extensions on DOS systems, a GLS locale file uses a condensed form of the code-set name, `codemodf`, in its file names.

The four-character `code` name of each locale file is the hexadecimal representation of the code-set number for the code set that the locale supports. The four-character `modf` name is the optional locale modifier.

For example, the ISO8859-1 code set has an IBM CCSID number of 819 in decimal and 0333 in hexadecimal. Therefore, the four-character name of a locale source file that supports the ISO8859-1 code set is `0333.lc`.

The following table shows some code sets and locale modifiers that IBM Informix products can support, along with their associated locale source file names.

Code set	Locale modifier	Locale source file
ISO8859-1 (IBM CCSID 819)	None	`0333.lc`
	Dictionary	`0333dict.lc`
Windows Code Page 1252 (West Europe)	None	`04e4.lc`
	Dictionary	`04e4dict.lc`
IBM CCSID 850	None	`0352.lc`
	Dictionary	`0352dict.lc`

A French locale that supports the ISO8859-1 code set has a GLS locale that is called `0333.lc` file in the **fr_fr** locale-file subdirectory. The default locale, U.S. English, also uses the ISO8859-1 code set (on UNIX platforms); a locale file that is called `0333.lc` is also in the **en_us** locale-file subdirectory. Because both the French and U.S. English locales support the Windows Code Page 1252, both the **fr_fr** and **en_us** locale-file subdirectories contain a `04e4.lc` locale file.

Other GLS files

In addition to GLS locale files, IBM Informix products might also use other GLS files.

These other files are:

- Code-set-conversion files map one code set to another.
- Code-set files define code-point values for code sets.
- The Windows `registry` file converts locale aliases to valid locale file names.

Code-set-conversion files

The *code-set-conversion file* describes how to map each character in a particular source code set to the characters of a particular target code set.

IBM Informix products can perform a given code-set conversion if code-set-conversion files exist to describe the mapping between the two code sets.

Important: A client application checks the code sets that the client and database locales support when it begins execution. If code sets are different, and no code-set-conversion files exist, the client application generates an error. For information, see "Establish a database connection" on page 1-24.

When an IBM Informix product needs to obtain code-set-conversion information, it accesses one of the GLS code-set-conversion files in the following table.

Platform	Code-set-conversion file
UNIX	`$INFORMIXDIR/gls/cvY/code1code2.cvo`
Windows	`%INFORMIXDIR%\gls\cvY\code1code2.cvo`

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM Informix product, `gls` is the subdirectory that contains the GLS files, and `Y` represents the version number for the code-set-conversion object-file format.

This rest of this section describes the remaining elements in the path name of GLS code-set-conversion files.

Code-set-conversion source and object files

Each code-set conversion file has two forms.

Each code-set-conversion file has the following two forms:

- The code-set-conversion *source* file is an ASCII file that describes the mapping to use for one direction of the code-set conversion.

  This file has a `.cv` extension and serves as documentation for the corresponding object file.

- The code-set-conversion *object* file is a compiled form of the code-set-conversion information.

  IBM Informix products use the object file to obtain code-set-conversion information quickly. Object code-set-conversion files have a `.cvo` file extension.

The header of the code-set-conversion source file (`.cv`) lists the two code sets that it converts and the direction of the conversion.

Code-set-conversion file names

To conform to DOS 8.3 naming conventions, GLS code-set-conversion files use a condensed form of the code-set names, `code1code2`, in their file names.

The eight-character name of each code-set-conversion file is derived from the hexadecimal representation of the code-set numbers of the source code set (`code1`) and the target code set (`code2`).

For example, the ISO8859-1 code set has an IBM CCSID number of 819 in decimal and 0333 in hexadecimal. The IBM CCSID 437 code set, a common IBM UNIX code set, has a hexadecimal value of 01b5. Therefore, the `033301b5.cv` code-set-conversion file describes the conversion from the CCSID 819 code set to the CCSID 437 code set.

Required for code-set conversion

IBM Informix products use the Code-Set Name-Mapping file to translate between code-set names and the more compact code-set numbers. You can use the `registry` file to find the hexadecimal values that correspond to code-set names or code-set numbers.

Most code-set conversion requires two code-set-conversion files. One file supports conversion of characters in code set A to their counterparts in code set B. Another supports the conversion in the return direction (from B to A). Such conversions are called *two-way* code-set conversions. For example, the code-set conversion between the CCSID 437 code set (hexadecimal 01b5 code number) and the CCSID 819 code set (or ISO8859-1 with a hexademical 0333 code number) requires the following two code-set-conversion files:

- The `01b50333.cv` file describes the mappings to use when IBM Informix products convert characters in the CCSID 437 code set to those in the ISO8859-1 code set.
- The `033301b5.cv` file describes the mappings to use when IBM Informix products convert characters in the ISO8859-1 code set to those in the CCSID 437 code set.

To be able to convert between these two code sets, an IBM Informix product must be able to locate both these code-set-conversion object files. Performing the conversion on only one direction would result in mismatched characters. For more information about mismatched characters, see “Perform code-set conversion” on page 1-29.

The following table shows some of the code-set conversions that IBM Informix products can support, along with their associated code-set-conversion source file names.

Source code set	Target code set	Code-set-conversion source file
ISO8859-1	Windows Code Page 1252	`033304e4.cvo`
Windows Code Page 1252	ISO8859-1	`04e40333.cvo`

Source code set	Target code set	Code-set-conversion source file
ISO8859-1	IBM CCSID 850	03330352.cvo
IBM CCSID 850	ISO8859-1	03520333.cvo
Windows Code Page 1252	IBM CCSID 850	04e40352.cvo
IBM CCSID 850	Windows Code Page 1252	035204e4.cvo

Code-set files

A *code-set file* (also called a character-mapping or *charmap* file) defines a code set for subsequent use by locale and code-set-conversion files.

A GLS locale includes the appropriate code-set file for the code set that it supports. In addition, IBM Informix products can perform code-set conversion between the code sets that have code-set files.

When an IBM Informix product needs to obtain code-set information, it accesses one of the GLS code-set files in the following table.

Platform	Code-set file
UNIX	$INFORMIXDIR/gls/cmZ/code.cmo
Windows	%INFORMIXDIR%\gls\cmZ\code.cmo

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM Informix product, gls is the subdirectory that contains the GLS files, and Z represents the version number for the code-set object-file format.

Each code-set file has the following two forms:

- The code-set *source* file is an ASCII file that describes the characters of a character set.

  This file has a .cm extension and serves as documentation for the corresponding object file.
- The code-set *object* file is a compiled form of the code-set information.

  The object file is used to create locale object files. Object code-set files have a .cmo file extension.

The IBM Informix registry file (Windows)

The Code-Set Name-Mapping file, which is called registry, is an ASCII file that associates code-set names and aliases with their code-set numbers.

A code-set number is based on the IBM CCSID numbering scheme. IBM Informix products use code-set numbers to determine the file names of locale and code-set-conversion files.

For example, you can specify the French locale that supports the ISO8859-1 code set with any of the following locale names as locale aliases:

- The full code-set name

  fr_fr.8859-1
- The decimal value of the IBM CCSID number

fr_fr.819

- The hexadecimal value of the IBM CCSID number

  fr_fr.0333

When you specify a locale name with either of the first two forms, IBM Informix products use the Code-Set Name-Mapping file to translate between code-set names (8859-1) or code-set number (819) to the condensed code-set name (0333). For information about the file format and search algorithm that IBM Informix products use to convert code-set names to code-set numbers, see the comments at the top of the registry file.

When an IBM Informix product needs to obtain information about locale aliases, it accesses the GLS code-set files in the following path:

%INFORMIXDIR%\gls\cmZ\registry

In these paths, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM Informix product, gls is the subdirectory that contains the GLS files, and *Z* represents the version number for the code-set object-file format.

Restriction: Do not remove the Code-Set Name-Mapping file, registry, from the IBM Informix directory. Do not modify this file. IBM Informix products use this file for the language processing of all locales.

Remove unused files

An IBM Informix product contains the GLS files.

These GLS files are:

- Locale files: source (*.lc) and object (*.lco)
- Code-set-conversion files: source (*.cv) and object (*.cvo)
- Code-set files: source only (*.cm)

Remove locale and code-set-conversion files

To save disk space, you might want to keep only those files that you intend to use.

These topics describe which of these files you can safely remove from your IBM Informix installation. You can safely remove the following GLS files from your IBM Informix installation:

- For those locales that you do not intend to use, you can remove locale source and object files (.lc and .lco) from the subdirectories of the lc*X* subdirectory in your IBM Informix installation.

  For more information about the lc*X* path name, see "Locale-file subdirectories" on page A-6.
- For those code-set conversions that you do not intend to use, you can remove code-set-conversion source and object files (.cv and .cvo) from the subdirectories of the cv*Y* subdirectory in your IBM Informix installation.

  For more information about the cv*Y* path name, see "Code-set-conversion file names" on page A-9.

Restriction: Do not remove the locale object file for the U.S. 8859-1 English locale, `0333.lco` in the **en_us** locale-file subdirectory. In addition, do not remove the Code-Set Name-Mapping file, `registry`. IBM Informix products use these files for the language processing of all locales.

Because IBM Informix products do not access source versions of locale and code-set conversion files, you can safely remove them. These files, however, provide useful online documentation for the supported locales and code-set conversions. If you have enough disk space, it is recommended that you keep these source files for the GLS locales (`*.lc`) and code-set conversions (`*.cv`) that your IBM Informix installation supports.

Remove code-set files

The source version of code-set files (`.cm`) are provided as online documentation for the locales and code-set conversions that use them. Because IBM Informix products do not access source code-set files, you can safely remove them.

However, if you have enough disk space, it is recommended that you keep these source files for the GLS locales and code-set conversions that your IBM Informix installation supports.

The glfiles utility (UNIX)

To comply with MS-DOS 8.3 legacy format for file names, IBM Informix products use condensed file names to store GLS locales and code-set-conversion files.

These file names do not match the names of the locales and code sets that the user uses. You can use the **glfiles** utility to generate a list of the following GLS-related files:

- The GLS locales that are available on your system
- The IBM Informix code-set-conversion files that are available on your system
- The IBM Informix code-set files that are available on your system

Before you run **glfiles**, take the following steps:

- Set the **INFORMIXDIR** environment variable to the directory in which you install your IBM Informix product.

  If you do not set **INFORMIXDIR**, **glfiles** checks the `/usr/informix` directory for the GLS files.
- Change to the directory where you want the files that **glfiles** generates.

  The utility creates the GLS file listings in the current directory.

The following diagram shows the syntax of the **glfiles** utility.

Element
Purpose

-cv The **glfiles** utility creates a file that lists the available code-set-conversion files.

-lc The **glfiles** utility creates a file that lists the available GLS locales.

-cm The **glfiles** utility creates a file that lists the available character mapping (charmap) files.

List code-set-conversion files

When you specify the **-cv** command-line option, the **glfiles** utility creates a file that lists the available code-set-conversion files.

For each cv*Y* subdirectory in $INFORMIXDIR/gls, **glfiles** creates a file in your current directory that is called cv*Y*.txt, where *Y* is the version number of the code-set-conversion object-file format. The cv*Y*.txt file lists the code-set conversions in alphabetical order, sorted on the name of the object code-set-conversion file.

For two-way code-set conversions, the $INFORMIXDIR/gls/cv*Y* directory contains two code-set-conversion files. One file supports conversion from the characters in code set A to their mappings in code set B, and another supports the conversion in the return direction (from code set B to code set A). For more information about two-way code-set conversion, see page "Code-set-conversion files" on page A-8.

The following figure shows a file, cv9.txt, that lists available code-set conversions.

```
Filenames: cv9/002501b5.cvo and cv9/01b50025.cvo
Between Code Set: Greek
    and Code Set: IBM CCSID 437

Filenames: cv9/00250333.cvo and cv9/03330025.cvo
Between Code Set: Greek
    and Code Set: ISO8859-1


Filenames: cv9/033304e4.cvo and cv9/004e40333.cvo
Between Code Set: 8859-1
    and Code Set: 1252
```

Figure A-1. Sample glfiles file for IBM Informix code-set-conversion files

Examine the cv*Y*.txt file to determine the code-set conversions that the $INFORMIXDIR/gls/cv*Y* directory on your system supports.

List GLS locale files

The **glfiles** utility can create a file that lists the available GLS locales.

The **glfiles** utility creates the file in the following ways:

- When you specify the **-lc** command-line option
- When you omit all command-line options

For each lc*X* subdirectory in the gls directory specified in **INFORMIXDIR**, **glfiles** creates a file in the current directory that is called lc*X*.txt, where *X* is the version number of the locale object-file format. The lc*X*.txt file lists the locales in alphabetical order, sorted on the name of the GLS locale object file.

The following figure shows a sample file, lc11.txt, that contains the available GLS locales.

```
Filename: lc11/ar_ae/0441.lco
Language: Arabic
Territory: United Arab Emirates
Code Set: 8859-6
Locale Name: ar_ae.1089

Filename: lc11/ar_ae/0441greg.lco
Language: Arabic
Territory: United Arab Emirates
Modifier: Gregorian
Code Set: 8859-6
Locale Name: ar_ae.1089@greg
. . .


Filename: lc11/en_us/0333.lco
Language: English
Territory: United States
Code Set: 8859-1
Locale Name: en_us.819

Filename: lc11/en_us/0333dict.lco
Language: English
Territory: United States
Code Set: 8859-1
Locale Name: en_us.819@dict

Filename: lc11/en_us/0352.lco
Language: English
Territory: United States
Code Set: PC-Latin-1
Locale Name: en_us.850

Filename: lc11/en_us/04e4.lco
Language: English
Territory: United States
Code Set: CP1252
Locale Name: en_us.1252
. . .
```

Figure A-2. Sample **glfiles** *file for GLS Locales*

Examine the lc*X*.txt files to determine the GLS locales that the $INFORMIXDIR/gls/lc*X* directory on your system supports.

To find out which GLS locales are available on your Windows system, you must look in the GLS system directories. A GLS locale resides in the file %INFORMIXDIR%\gls\lc*X*\lg_tr*codemodf*.lco

In this path, **INFORMIXDIR** is the environment variable that specifies the directory in which you install the IBM Informix product, gls is the subdirectory that contains the GLS system files, *X* represents the version number of the locale file format, lg is the two-character language name, tr is the two-character territory name that the locale supports, and *codemodf* is the condensed locale name.

List character-mapping files

When you specify the **-cm** command-line option, the **glfiles** utility creates a file that lists the available character mapping (charmap) files.

For each cm*Z* subdirectory in $INFORMIXDIR/gls, **glfiles** creates a file in the current directory that is called cm*Z*.txt, where *Z* is the version number of the charmap

object-file format. The cm*Z*.txt file lists the character mappings in alphabetical order, sorted on the name of the GLS object charmap file.

Figure A-3 shows a sample file, cm3.txt, that contains the available character mappings.

```
Filename: cm3/032d.cm
Code Set: 8859-7

Filename: cm3/0333.cm
Code Set: 8859-1

Filename: cm3/0352.cm
Code Set:  PC-Latin-1

Filename: cm3/04e4.cm
Code Set:  CP1252
```

Figure A-3. Sample **glfiles** *file for IBM Informix character-mapping files*

Examine the cm*Z*.txt file to determine the character mappings that the $INFORMIXDIR/gls/cm*Z* directory on your system supports.

Appendix B. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix products

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

IBM and accessibility

For more information about the IBM commitment to accessibility, see the *IBM Accessibility Center* at http://www.ibm.com/able.

Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

© Copyright IBM Corp. 1996, 2013

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 refers to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be

repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or you can include none. If you hear the lines `3*, 3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The `*` symbol is equivalent to a loop-back line in a railroad syntax diagram.

`+` Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line `6.1+ data-area`, you must include at least one data area. If you hear the lines `2+`, `2 HOST`, and `2 STATE`, you know that you must include `HOST`, `STATE`, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the `*` symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

© Copyright IBM Corp. 1996, 2013

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy,

modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

_ (underscore), wildcard of LIKE operator 3-26
- (minus sign), wildcard in MATCHES clause 3-26
? (question mark), wildcard of MATCHES operator 3-26
.c file extension 5-6, 6-4
.c_ file extension 6-4
.cm file extension A-10, A-12
.cmo file extension A-10
.cv file extension 1-29, A-8, A-11
.cvo file extension A-8, A-11
.ec file extension 5-6, 6-4
.iem file extension 2-4
.lc file extension A-6, A-8, A-11
.lco file extension A-6, A-11
.o file extension 6-4
* (asterisk), wildcard of MATCHES operator 3-26
[ ] (brackets)
  ranges with MATCHES operator 3-24, 3-26
  substring operator 3-13
@ (at sign)
  as formatting character 6-12
% (percent)
  formatting directive 2-10, A-2
  in parameter markers 4-13
  in trace messages 4-13, 4-14
  wildcard of LIKE operator 3-26
^ (caret), wildcard in MATCHES clause 3-26

Numerics

1-based counts 1-26
8-bit clean 1-9, 2-7, 6-3

A

Abbreviations 1-15
Accessibility B-1
  dotted decimal format of syntax diagrams B-1
  keyboard B-1
  shortcut keys B-1
  syntax diagrams, reading in a screen reader B-1
Alias 1-9, 3-2
Alpha class 3-5
Alphabetic characters 1-10, 3-5
ALTER TABLE statement 3-31
ALTER TABLE statements
  character column declarations 3-36
Alternative formats
  date 2-12, 6-8
  time 2-17, 6-10
ANSI compliance
  comment indicators 3-13
  owner naming 3-3
  quotation marks 3-4
ASCII code set 1-9, 1-22
ASCII letters (a - z, A - Z) 3-5
Asian language support (ALS) 1-5
Asterisk (*) symbol, with MATCHES operator 3-26
Authorization identifier 3-3

B

Basic Multilingual Plane (BMP) 1-30
BETWEEN conditions 3-22
BLOB data type, searching in 3-33
Bracket ([ ]) symbols
  ranges for MATCHES operator 3-24
  substring operator 3-13
BYTE data type
  code-set conversion 5-3, 6-14
  partial characters 3-17

C

C compiler
  8-bit clean 4-6, 6-3
  limitations 4-6, 6-3
  multibyte characters 4-6, 6-3
  non-ASCII filenames 6-3
  non-ASCII source code 4-6, 6-3
Case-insensitive databases 3-6, 3-9
Casts 3-2
CC8BITLEVEL environment variable 1-3, 2-2, 6-3, 6-4
CHAR data type
  and GLS 1-7
  code-set conversion 5-3
  collation order 1-13
  difference from NCHAR 3-6
  GLS aspects 3-10
CHAR_LENGTH function 3-30
Character
  7-bit 1-9
  8-bit 1-9
  ASCII 1-9
  mismatched 1-29, A-9
  nonprintable 3-8, 3-10
  partial 3-15, 5-7
  shifting lettercase 6-14
  single-byte 1-9, 3-14
Character data
  avoiding corruption of 5-3
  collation of 1-27, 3-18, A-4
  converting 1-29, 5-3
  data types 3-6
  equivalent characters 1-12, 3-19, 3-23, A-4
  ESQL functions 6-14
  interpreting 1-19, 1-27
  mapping A-1
  processing with locales 1-6
Character set 1-9, A-10
CHARACTER_LENGTH function 3-27
Character-mapping files A-14
Chinese locale 1-24
chkenv utility 4-4
Chunks 3-2
Client application
  checking a connection 1-25, 1-27, 5-1
  code-set conversion 5-1
  definition of 1-6
  end-user formats 1-13
  establishing a connection 5-1

© Copyright IBM Corp. 1996, 2013

Client application *(continued)*
opening another database 1-27, 5-1
requesting a connection 1-19, 1-24
sending client locale to server 1-25, 1-28
setting a locale 1-9, 1-18, 1-23
support for locales 1-6, 1-8
uses of client locale 1-18
verifying locales 5-1
Client code set 1-29, 5-1
Client computer
client code set 1-29
code-set-conversion files 5-1
setting CLIENT_LOCALE 1-23
setting DB_LOCALE 1-23
Client locale
code set 1-29, 5-1
COLLATION category 1-18
CTYPE category 1-18
customizing 1-32
definition of 1-18
determining 1-18
ESQL/C source files 6-1
MESSAGES category A-6
MONETARY category 1-18
NUMERIC category 1-18
sample 1-18
sending to database server 1-25
setting 1-23
TIME category 1-18
CLIENT_LOCALE environment variable 1-3
default value 1-23
ESQL file names 5-6
ESQL source code 5-6
example of locale name 2-2
interpreting command-line arguments 4-4
location of message files 2-4
precedence of 1-18, 1-28, 1-33, 1-34, 2-4, 6-9, 6-10, 6-11
role in code-set conversion 4-2, 5-1
role in exception messages 4-10
sending to database server 1-25
setting 1-23
syntax 2-2
with TEXT data 3-7, 3-9, 3-10, 3-11, 3-12
with utilities 4-3
Client/server environment
client locale 1-18, 1-25
code-set conversion 1-29, 1-30
database locale 1-19
locales of 1-9, 1-17
server locale 1-21
server-processing locale 1-26
setting environment variables 1-23
cmZ.txt file A-14
Code points 1-9, 1-11, 3-10
Code sets
1252 1-9
8859-1 1-9, 1-22, A-7
affecting filenames 2-8
ASCII 1-9, 1-22
character classes 1-10
client code set 1-29
code points 1-9, 3-10
compatible 1-6
condensed name 1-19, 1-22, 1-32, A-7
convertible 1-23, 5-1, 5-3
database code set 1-29
default 1-10, 1-22, 1-24
Code sets *(continued)*
definition of 1-9
determining 1-24, 1-29
for client applications 1-29, 5-1
for database 1-29, 5-1
for database server 1-29, 5-1
GB18030-2000 1-9
in locale name 1-22, 1-25, 2-5, 2-21
incompatible 5-1
multibyte 1-9, 3-14, 3-15, 3-27, 5-7
server code set 1-29
single-byte 1-9, 3-14, 3-16, 3-27
source 1-29
target 1-29
UTF-8 1-13
wide-character form 4-7
Code-set conversion
by client application 5-1
by database server 4-2
by DataBlade API 4-8
character mismatches 1-29, A-9
data converted 5-3
definition of 1-29
for column names 5-3
for cursor names 5-3
for error message text 5-3
for LVARCHAR 5-3
for opaque types 4-8
for simple-large-object data 5-3, 6-15
for SQL data types 5-3
for SQL statements 5-3
for statement IDs 5-3
for table names 5-3
globalized error messages and 4-10
handling mismatched characters 1-29
in ESQL/C program 6-14
limitations 1-29
lossy error 1-29
performing 1-30, 4-2, 5-3
registry file A-10, A-11
role of CLIENT_LOCALE 4-2, 5-1
role of DB_LOCALE 4-2, 5-1
role of SERVER_LOCALE 4-2
two-way A-9
Code-set file
description of 1-9, A-10
listing A-14
location of A-10
object A-10
removing A-12
source A-10
Code-set-conversion file
description of 1-9, A-8
listing 5-1, A-13
location of A-8
object A-8, A-11
removing unused A-11
source 1-29, A-8, A-11
Collation
definition of 1-11
equivalence classes 1-12, 3-19, 3-23, 3-24, 3-26, A-4
of character data 3-18
of NCHAR 3-6
of NVARCHAR 3-9
Unicode collation 1-13
weights A-4

COLLATION locale category
description of A-2, A-4
in client locale 1-18
in locale source file A-6
in server-processing locale 1-27
Collation order
code-set 1-11, 1-13, 3-10
localized 1-6, 1-12, 1-13, 1-27, 2-2, 2-5, 2-21, 3-10
tasks affected by 1-11
types of 1-11
Column (database)
expressions 3-13
in code-set conversion 5-3
naming 1-6, 3-2, 4-6, 6-1
substrings 3-13, 3-17
Command-line
arguments 4-4
Comment indicators 3-13
Comments 2-2, 3-13, 6-1
compliance with standards x
Conditions
BETWEEN 3-22
IN 3-23
LIKE 3-26
MATCHES 3-24
relational operator 3-21
Configuration parameters 4-1
SQL_LOGICAL_CHAR 3-36
CONNECT statement 3-2
Constraints 3-2, 4-6, 6-1
Conversion functions 5-6
Conversion modifier 1-33, 2-12, 2-17
CREATE CAST statement 3-2
CREATE DATABASE statement 3-2, 3-5
CREATE DISTINCT TYPE statement 3-2
CREATE FUNCTION statement 3-2
CREATE INDEX statement 3-1, 3-2, 3-18
CREATE OPAQUE TYPE statement 3-2
CREATE OPCLASS statement 3-2
CREATE PROCEDURE statement 3-2
CREATE ROLE statement 3-2
CREATE ROW TYPE statement 3-2
CREATE SEQUENCE statement 3-2
CREATE SYNONYM statement 3-2
CREATE TABLE statement 3-5
column name in 3-2
constraint name in 3-2
MONEY columns 3-31
names of database objects 3-1
table name in 3-2
CREATE TABLE statements
character column declarations 3-36
CREATE TRIGGER statement 3-2
CREATE VIEW statement 3-2, 3-5
CTYPE locale category
character case 6-14
description of A-2, A-3
in client locale 1-18
in locale source file A-6
in server-processing locale 1-27
white space characters 2-10, 2-15
Currency notation 1-15, 1-34, 2-5
Currency symbol 1-15, 1-22, 3-32, 6-11, A-5
Current processing locale 4-5, 4-12
Cursor 1-6, 3-2, 4-6, 5-3, 6-1
cvY.txt file A-13
Cyrillic alphabet 3-5

D

Data
character 3-6
converting 5-3
corruption 1-18, 1-19
transferring 1-26
Data definition language (DDL) 3-1
Data types
BLOB 3-33
BYTE 5-3
CHAR 3-10, 5-3
character 3-6
CHARACTER 3-36
CHARACTER VARYING 3-36
CLOB 3-33, 3-36
code-set conversion of 5-3
collation order of 1-13
collection 3-36
complex 3-33
DATE A-5
DATETIME A-5
DECIMAL A-4
distinct 3-33, 3-36
FLOAT A-4
INTEGER A-4
internal format 1-13
LIST 3-36
locale-sensitive 1-19, 1-27, 3-6, 3-31, 6-6
locator structure 6-14
LVARCHAR 3-11, 3-36, 4-16
MULTISET 3-36
NCHAR 1-7, 3-6, 5-3, 6-6
numeric A-4
NVARCHAR 1-7, 3-8, 3-36, 5-3, 6-6
opaque 3-33, 4-8, 4-16
ROW 3-36
SET 3-36
SMALLFLOAT A-4
SMALLINT A-4
storage size declarations 3-36
TEXT 3-12, 3-36, 5-3
user-defined (UDTs) 3-36
VARCHAR 3-11, 3-36, 5-3
Database code set 1-29, 3-36, 5-1
Database locale
code set 1-29, 5-1
definition of 1-19
for UDR trace messages 4-13
in system catalog 1-19, 1-25
incompatible 1-25
setting 1-23
verifying 1-19, 1-25, 1-27
Database objects
and DB-Access 1-6
naming 3-1
Database server connection
client-locale information 1-25
establishing 1-24, 5-1
example 1-19
naming 3-2
sample 1-18
server-processing locale 1-18
verifying 1-24, 1-25, 1-27, 5-1
warnings 1-25, 1-26
Database servers
chunk name 3-2
code-set conversion 1-30, 4-2

Database servers *(continued)*
collation 1-13
determining server-processing locale 1-24, 1-26
diagnostic files 4-1
end-user formats 1-13
internal formats 1-13
interpreting character data 1-19
log filename 3-2
message log file 4-1
multibyte characters 4-3
multibyte filenames 3-2
operating-system files 4-1
sample connection 1-17
setting a locale 1-9, 1-23
support for locales 1-5, 1-8
uses of client locale 1-24, 1-25
uses of server locale 1-21, 4-1
using DB_LOCALE 1-19
utilities 1-6, 4-3
verifying a connection 1-24, 5-1
verifying database locale 1-25, 1-27
Databases
loading 3-35
naming 3-2, 4-6, 6-1
saving locale information 1-19
unloading 3-36
DataBlade Developers Kit (DBDK) 4-8
Date data
alternative formats 2-12
customizing format of 1-32
end-user format 1-22, 1-28, 1-32, A-5
format of A-5
locale-specific 1-6, 1-13
precedence of environment variables 1-33, 6-9
DATE data
setting GL_DATE 2-10
DATE data type
end-user format 1-22, 1-32, 2-3, 2-10, A-5
era-based dates 1-33
ESQL library functions 6-7
extended-format strings 6-8
internal format 1-13, 1-15
precedence of environment variables 1-33, 6-9
DATETIME data type
end-user format 1-22, 1-32, 2-6, 2-15, A-5
era-based dates 1-33
ESQL library functions 6-9
extended-format strings 6-10
formatting directives for 2-15
internal format 1-15
precedence of environment variables 1-33, 6-10
DB_LOCALE environment variable 1-3
default value 1-23
example of locale name 2-5
information it determines 1-19, 1-21
precedence of 1-27
role in code-set conversion 4-2, 5-1
role in exception messages 4-10
setting 1-23
syntax 2-5
verifying database locale 1-25
with utilities 4-3
DB-Access utility 1-6, 4-4
DBCENTURY environment variable 2-11
DBDATE environment variable
era-based dates 1-33, 3-34
ESQL library functions 6-7
DBDATE environment variable *(continued)*
precedence of 1-18, 1-28, 1-33, 6-9
setting 1-32
syntax 2-3
dbexport utility 1-6, 2-7, 4-4
dbimport utility 4-4
DBLANG environment variable 1-3, 5-5
precedence of 2-4
setting 1-32
syntax 2-4
dbload utility 4-4
DBMONEY environment variable 1-3
defining currency symbols 6-12
ESQL library functions 6-11, 6-13
precedence of 1-18, 1-28, 1-34, 6-11
sending to database server 1-25
setting 1-34
syntax 2-5
DBNLS environment variable 3-7, 5-1
dbschema utility 4-4
dbspaces
Unicode 1-13
DBTIME environment variable 1-3
era-based dates 3-34
ESQL library functions 6-10
precedence of 1-18, 1-28, 1-33, 6-10
setting 1-32
syntax 2-6
DECIMAL data type 1-34, A-4
Decimal separator 1-15, 1-22, 3-32, 6-11, A-2, A-4, A-5
DECLARE statement 3-2
Default locale
default code set 1-22, 1-24, A-7
for client application 1-23
for database server 1-23
locale name 1-22
required A-11
DEFINE statements of SPL 3-36
DELETE statements
era-based dates 3-34
GLS considerations 3-34
WHERE clause conditions 3-34
DELIMIDENT environment variable 3-4, 3-12
DESCRIBE statement 6-16
Diagnostic file 1-21, 4-1
Disabilities, visual
reading syntax diagrams B-1
Disability B-1
Distinct data types 3-2
Dollar ($) sign
as formatting character 6-12
currency symbol A-2
in identifiers 3-1
Dotted decimal format of syntax diagrams B-1
Double (") quotes 3-4
dtcvfmasc() library function 6-9
dttofmtasc() library function 6-9
DUMP* configuration parameters 4-1

E

End-user format
conversion modifier 2-12, 2-17
customizing 1-32
date data 1-15, 1-22, 1-32, 4-9, A-5
DATE data 2-10, 2-15
date format qualifiers 2-13

End-user format *(continued)*
DATETIME data 2-15
default 1-22
definition of 1-13, 1-32, 1-34
environment variables 1-13
extended DATE-format strings 6-8
extended DATETIME format strings 6-10
formatting data 4-9, 5-6
locale categories 1-13
monetary data 1-15, 1-22, 1-34, 2-5, 4-9, A-5
numeric data 1-15, 1-22, 4-9, A-4
printing 1-15, 2-14, 2-18
scanning 1-15, 2-18
sending to database server 1-25, 1-28
time data 1-15, 1-22, 1-32, A-5
time format qualifiers 2-18
English locale 1-24, A-6
Environment variables
CC8BITLEVEL 2-2, 6-4
CLIENT_LOCALE 1-23, 2-2
DB_LOCALE 1-23, 2-5
DBCENTURY 2-11
DBDATE 2-3
DBLANG 2-4, 5-5
DBMONEY 2-5
DBNLS 5-1
DBTIME 2-6, 2-15
DELIMIDENT 3-4, 3-12
ESQLMF 2-7, 6-4
for end-user formats 1-13
GL_DATE 2-10
GL_DATETIME 2-15
GL_PATH 2-1
GL_USEGLU 2-20
GLS-related 2-1
GLS8BITFSYS 2-7
INFORMIXDIR 5-5
locale 4-3
locale-related 1-23
precedence for client locale 1-18
precedence for DATE data 1-33, 6-9
precedence for DATETIME data 1-33, 6-10
precedence for monetary data 1-34, 6-11
precedence for server-processing locale 1-27, 1-28
SERVER_LOCALE 1-23, 2-21
USE_DTENV 2-15, 2-20
Era-based dates
DATE-format functions 6-7
DATETIME-format functions 6-9
DBDATE formats 6-7
DBTIME formats 6-10
defined in locale A-5
definition of 1-15
extended-format strings 6-8, 6-10
GL_DATE formats 1-33, 2-12
GL_DATETIME formats 1-33
in DELETE statement 3-34
in INSERT statement 3-34
in SQL statements 3-34
in UPDATE statement 3-34
sample 1-15
Error message files 5-5
Error messages
DATE-format 6-14
DATETIME-format 6-14
globalizing 4-10
GLS-specific 6-14
Error messages *(continued)*
in code-set conversion 5-3
numeric-format 6-14
Escape character 3-26
ESQL library functions
currency notation in 6-11, 6-12
DATE-format functions 6-7
DATETIME-format functions 6-9
GLS enhancements 6-7
numeric-format functions 6-11
string functions 6-14
ESQL/C data types 1-7, 5-3, 6-6
ESQL/C filter
description of 6-3
invoking 6-4
non-ASCII characters 6-3
with CC8BITLEVEL 6-4
with CC8BITLEVEL environment variable 2-2
with ESQLMF 2-7, 6-4
ESQL/C function library
dtcvfmtasc() 6-9
dttofmtasc() 6-9
GLS error messages 6-14
precedence for DATE data 6-9
precedence for DATETIME data 6-10
precedence for MONEY data 6-11
rdatestr() 6-7
rdefmtdate() 6-7, 6-8
rdownshift() 6-14
rfmtdate() 6-7, 6-8
rfmtdec() 6-11
rfmtdouble() 6-11
rfmtlong() 6-11
rstrdate() 6-7
rupshift() 6-14
ESQL/C preprocessor 1-18, 6-3
ESQL/C processor
definition of 5-6
invoking ESQL/C filter 2-2, 6-4
multibyte characters 2-7, 6-3
non-ASCII file names 2-7
non-ASCII filenames 6-3
non-ASCII source code 6-4
operating-system files 5-6
with CC8BITLEVEL 2-2
with ESQLMF 2-7, 6-4
ESQL/C program
accessing NCHAR data 6-6
accessing NVARCHAR data 6-6
checking database connection 1-25, 1-26
comments 2-2, 6-1
compiling 6-4
data type constants 6-15
filenames 6-1
handling code-set conversion 6-14
host variables 1-18, 6-1
indicator variables 6-1
literal strings 1-13, 1-18, 2-2, 6-1
writing simple large objects to database 6-14
ESQLMF environment variable 1-3, 2-7, 6-4
Explain file 1-21
External representation of opaque data 4-16

F

FETCH statement 3-2

File extensions
.c 5-6, 6-4
.c_ 6-4
.cm A-10, A-12
.cmo A-10
.cv 1-29, A-8, A-11
.cvo A-8, A-11
.ec 5-6, 6-4
.iem 2-4
.lc A-6, A-8, A-11
.lco A-6, A-11
.o 6-4
Filename
7-bit clean 2-7
8-bit clean 1-9
generating 2-8, 6-3
illegal characters in 2-7
non-ASCII 2-8, 3-2, 4-6, 6-1, 6-3
validating 4-2
Files
cmZ.txt A-14
cvY.txt A-13
diagnostic 1-21, 4-1
explain output file 1-21
Informix-proprietary 1-21
lcX.txt A-13
LOAD FROM 3-35
locale object file A-6
locale source file A-6
log 1-21, 4-1
message 1-21, 1-30, 1-32, 2-4
registry 1-9, A-10, A-11
text 3-35
UNLOAD TO 3-36
finderr utility 6-14
FLOAT data type 1-34, A-4
Formatting 5-6
Formatting directive
conversion modifiers 1-33, 2-12
field precision 2-14, 2-18
field specification 2-13, 2-14, 2-18
field width 2-14, 2-18
white space 2-10
with GL_DATE 2-10
with GL_DATETIME 2-15
French locale 1-13, 1-15, 1-24, 1-27, 2-2, 2-5, 2-21, 3-5, 5-1, A-6
Functions, case-sensitive 3-18

G

GB18030-2000 code set 1-9, 1-30, 2-20
Gengo year format 1-15
German locale 1-18, 1-19, 1-24, A-6
GL_DATE environment variable 1-3
era-based dates 1-33, 3-34
ESQL library functions 6-7
formatting directives 2-10
precedence of 1-18, 1-28, 1-33, 6-9
sending to database server 1-25
setting 1-32
syntax 2-10
GL_DATETIME environment variable 1-3
era-based dates 3-34
era-based dates and times 1-33
ESQL library functions 6-9
formatting directives 2-15
precedence of 1-18, 1-28, 1-33, 6-10
GL_DATETIME environment variable *(continued)*
sending to database server 1-25
setting 1-32
syntax 2-15
gl_dprintf() function 4-13
GL_DPRINTF() tracing function 4-14
GL_PATH environment variable 2-1
gl_tprintf() function 4-13
gl_tprintf() tracing function 4-14
GL_USEGLU environment variable 2-20
gl_wchar_t data type 4-7
glfiles utility
-cm option A-14
-cv option A-13
-lc option A-13
charmap files A-14
code-set files A-14
code-set-conversion files 5-1, A-13
locale files 2-2, 2-5, 2-21, A-13
sample output A-13, A-14
syntax A-12
Globalization
C UDRs and 4-5
definition of 5-3
formatting data 4-9, 5-6
of error messages 4-10
of trace messages 4-13
processing characters 4-7, 5-6
UDRs and 4-5
GLS feature
available locales 2-2, 2-5, 2-21
CHAR data type 3-10
character data types for host variables 6-6
client/server environment 1-9, 1-17
description of 1-1
environment variables 2-1
ESQL library functions 6-7
for DataBlade modules 1-7
for SQL 3-1
functionality listed 1-5
fundamentals 1-1
GLS files A-6, A-8, A-10
GLS library 1-3
managing GLS files A-1
NCHAR data type 3-6
NVARCHAR data type 3-8
TEXT data type 3-12
using character data types 3-6
VARCHAR data type 3-11
GLS locale file 1-9
GLS_COLLATE tabname 1-19
GLS_CTYPE tabname 1-19
GLS8BITFSYS environment variable 1-3, 2-7
Graphical-replacement conversion 1-29
Greek alphabet 3-5
Gregorian calendar 1-15

H

Heisei era 1-15
HKEY_LOCAL_MACHINE registry setting 2-21, 4-3
Host variable
end-user formats 1-13
ESQL/C example 6-1, 6-2
naming 1-6, 3-2, 6-1, 6-2

I

IBM CCSID code set
  437 1-30, A-9
  819 A-7, A-9, A-10
  definition of 1-30
IBM Informix Client Software Development Kit 5-1
IBM Informix Dynamic Server, pathnames 3-2
IBM Informix GLS API 1-7, 4-7
Identifier
  delimited 3-2
  Non-ASCII characters 3-2
IN conditions 3-23
Index 3-2
Index keys
  Unicode 1-13
Indicator variable 1-6, 6-1, 6-2
industry standards x
INFORMIXDIR environment variable 5-5
  location of charmap files A-14
  location of code-set files A-10, A-14
  location of code-set-conversion files A-8, A-13
  location of locale files 1-16, A-6, A-13
  location of message files 2-4
  location of registry file A-10
  with glfiles A-12
INITCAP function 3-12, 3-18
INSERT statements
  embedded SELECT 3-34
  end-user formats 1-13
  era-based dates 3-34
  GLS considerations 3-34
  specifying quoted strings 3-12
  VALUES clause 3-34
INTEGER data type A-4
International Components for Unicode (ICU) 1-9
International Language Supplement 1-8
International Language Supplement (ILS) 1-8

J

ja_jp.sjis locale 6-7
Japanese Imperial dates 1-15, 1-33
Japanese locale 1-23, 1-24, 1-27, 5-1
Japanese UJIS locale 3-5
Join condition 3-21

K

Kanji characters 3-5
Korean locale 1-24

L

LANG environment variable
  precedence of 2-4
Language
  code sets 1-30
  default 1-22
  for client application 1-18
  for database 1-19
  for database server 1-21
  in locale name 1-25, 2-5, 2-21, A-6
lcX.txt file A-13
Left-to-right writing direction 1-19
LENGTH function 3-27
LIBMI applications 1-7
LIKE relational operator 1-11, 3-26
Literal matches 3-24, 3-26
Literal string 1-13, 2-2, 4-6, 6-1
Load file 3-35
LOAD statement 3-2, 3-34, 3-35
loc_buffer field 6-15
loc_t data type 6-14, 6-15
loc_type field 6-15
Locale environment variables 4-3
Locale file
  description of 1-9, 1-16, A-2
  listing 2-2, 2-5, 2-21, A-12, A-13
  location of 1-16, A-6
  object A-6, A-11
  removing unused A-11
  required A-11
  source A-6, A-11
Locale modifier 1-25, 2-2, 2-5, 2-21, A-7
Locale name 2-2
  code-set name 1-22, 1-25, 2-2, 2-5, 2-21
  example 2-2, 2-5, 2-21
  language name 1-25, 2-2, 2-5, 2-21, A-6
  locale modifier name 1-25, 2-2, 2-5, 2-21, A-7
  territory name 1-25, 2-2, 2-5, 2-21, A-6
Locale-sensitive data types 1-26
Locales
  alpha class 3-5
  character classes 1-10
  choosing 5-4
  current 5-4
  current processing 4-5, 4-12
  definition of 1-9
  environment variables 1-23
  file name A-7
  filename A-6
  for database server connections 1-24
  in custom messages 4-12
  in trace messages 4-15
  listing 2-2, 2-5, 2-21, A-12
  locale categories 1-13, A-2
  non-ASCII characters 1-24
  setting 1-16, 1-23
  verifying 1-25, 1-27
Localization 5-4
Locator structure 6-15
Log file name, non-ASCII characters in 3-2
Log files 1-21, 4-1
Logical character semantics in type declarations 3-36
Lossy error 1-29
Lower class 1-10
LOWER function 3-12, 3-18
LVARCHAR data type
  code-set conversion 5-3
  collation order 1-13
  GLS aspects 3-11
  multibyte characters 3-36
  representing opaque data types 4-16

M

malloc() system call 6-16
MATCHES relational operator 1-11, 3-24
MERGE statement 3-34
MERGE statements
  embedded SELECT 3-34
  SET clause 3-34

MERGE statements *(continued)*
VALUES clause 3-34
WHERE clause conditions 3-34
Message file
compiled 2-4
language-specific 2-4
localized 1-32
locating at runtime 2-4
requirements 5-5
specifying location of 1-32, 2-4
Message log
and code-set conversion 1-30
non-ASCII characters in 2-9
MESSAGES locale category
description of A-2, A-6
in locale source file A-6
in server-processing locale 1-28
mi_date_to_string() DataBlade API function 4-9
mi_datetime_to_string() function 4-9
mi_db_error_raise() function 4-10
mi_decimal_to_string() DataBlade API function 4-9
mi_exec_prepared_statement() function 4-5
mi_exec() function 4-5, 4-6, 4-11
mi_get_string() DataBlade API function 4-8
mi_interval_to_string() function 4-9
MI_LIST_END tracing constant 4-14
mi_money_to_string() DataBlade API function 4-9
mi_prepare() function 4-6
mi_put_string() DataBlade API function 4-8
mi_string_to_date() DataBlade API function 4-9
mi_string_to_datetime() function 4-9
mi_string_to_decimal() DataBlade API function 4-9
mi_string_to_interval() function 4-9
mi_string_to_money() DataBlade API function 4-9
mi_wchar data type 4-7
Ming Guo year format 1-15, 1-33
Minus (-) sign
unary operator A-2
Monetary data
currency notation 1-13, 3-32, A-5
currency symbol 1-15, 1-22, 3-32, 6-11, A-5
decimal separator 1-15, 1-22, 3-32, 6-11, A-5
default scale 3-31
end-user format 1-22, 1-28, 1-34, A-5
format of A-5
locale-specific 1-6
negative 1-15, 1-22, A-5
positive 1-15, 1-22, A-5
precedence of environment variables 1-34, 6-11
thousands separator 1-15, 1-22, 3-32, 6-11, A-5
MONETARY locale category
currency symbol 6-12
description of A-2, A-5
end-user formats A-5
in client locale 1-18
in locale source file A-6
in server-processing locale 1-28
numeric-formatting functions 6-11
MONEY data type
defining 3-31
end-user format 2-5
internal format 1-15, 1-34, 3-31
precedence of environment variables 1-34, 6-11
MSGPATH configuration parameter 2-9, 4-1
Multibyte characters 4-7
column substrings 3-14
definition of 1-9
Multibyte characters *(continued)*
filtering 6-3
in cast names 3-2
in column names 1-6, 3-2, 4-6, 6-1
in comments 2-2, 6-1
in connection names 3-2
in constraint names 3-2, 4-6, 6-1
in cursor names 1-6, 3-2, 4-6, 6-1
in data type names 3-2
in database names 3-2, 4-6, 6-1
in database server file names 3-2
in database server filenames 3-2
in database server utilities 4-3
in delimited identifiers 3-2
in ESQL file names 6-3
in file names 2-8, 3-2, 4-6, 6-1
in filenames 1-24
in function names 3-2
in host variables 1-6, 3-2, 6-1, 6-2
in index names 3-2
in indicator variables 1-6, 6-1
in literal strings 2-2, 4-6, 6-1
in LOAD FROM file 3-35
in NCHAR columns 3-7
in numeric formats 6-11
in NVARCHAR columns 3-9
in opaque data type names 3-2
in operator-class names 3-2
in owner names 3-3
in procedure names 3-2
in quoted strings 3-12
in role names 3-2
in routine names 3-2
in ROW data type names 3-2
in sequence names 3-2
in SPL routines 1-6, 3-2
in SQL comments 3-13
in statement IDs 1-6, 3-2, 4-6, 6-1
in synonym names 3-2
in table aliases 3-2
in table names 1-6, 3-2, 4-6, 6-1
in trigger names 3-2
in triggers 3-2
in UNLOAD TO file 3-36
in view names 1-6, 3-2, 4-6, 6-1
partial characters 3-15, 5-7
processing 2-2, 5-6, 6-3
shifting case of 6-14
storage requirements 3-36
support by C compiler 4-6, 6-3
support for 1-24
with CC8BITLEVEL environment variable 2-2
with GLS8BITFSYS environment variable 2-8
with SQL_LOGICAL_CHAR configuration parameter 3-36
Multicharacter collation elements A-4

N

National language support (NLS) 1-5
NCHAR data type
code-set conversion 1-7, 5-3
collation order 1-13, 3-6
description of 3-6
difference from CHAR 3-6
in ESQL/C program 6-6
in regular expressions 1-6
inserting into database 6-6

NCHAR data type *(continued)*
multibyte characters 3-7
nonprintable characters 3-8
with numeric values 3-7
NLSCASE INSENSITIVE database property 3-6, 3-9
Non-ASCII character
definition of 1-9
examples 1-24
filtering 6-3
in cast names 3-2
in column names 1-6, 3-2, 4-6, 6-1
in comments 2-2, 6-1
in connection names 3-2
in constraint names 3-2, 4-6, 6-1
in cursor names 1-6, 3-2, 4-6, 6-1
in database names 3-2, 4-6, 6-1
in delimited identifiers 3-2
in distinct data type names 3-2
in ESQL filenames 6-3
in file names 2-8
in filenames 3-2, 4-6, 6-1
in host variables 1-6, 3-2, 6-1, 6-2
in index names 3-2
in indicator variables 1-6, 6-1
in literal strings 2-2, 4-6, 6-1
in LOAD FROM file 3-35
in opaque data type names 3-2
in operator-class names 3-2
in owner names 3-3
in quoted strings 3-12
in role names 3-2
in ROW data type names 3-2
in sequence names 3-2
in SPL routines 1-6, 3-2
in SQL comments 3-13
in statement IDs 1-6, 3-2, 4-6, 6-1
in synonym names 3-2
in table names 1-6, 3-2, 4-6, 6-1
in trigger names 3-2
in triggers 3-2
in UDR source files 4-5
in UNLOAD TO file 3-36
in view names 1-6, 3-2, 4-6, 6-1
processing 2-2, 6-3
support for 1-24
with CC8BITLEVEL environment variable 2-2
with GLS8BITFSYS environment variable 2-8
Non-Gregorian calendar 1-15
Non-Roman alphabets 3-5
Nondefault page size and Unicode 1-13
Numeric data
currency notation in 6-11
decimal separator 1-15, 1-22, 6-11, A-4
end-user format 1-13, 1-22, 1-28, A-4
ESQL functions 6-11
format of A-4
locale-specific 1-6
negative 1-15, 1-22, A-4
positive 1-15, 1-22, A-4
thousands separator 1-15, 1-22, 6-11, A-4
NUMERIC locale category
alternative digits 2-12, 2-17, A-4
description of A-2, A-4
end-user formats A-4
in client locale 1-18
in locale source file A-6
in server-processing locale 1-28
NUMERIC locale category *(continued)*
numeric-formatting functions 6-11
Numeric notation 1-15
NVARCHAR data type
code-set conversion 1-7, 5-3
collation order 1-13, 3-9
description of 3-8
difference from VARCHAR 3-9
in ESQL/C program 6-6
in regular expressions 1-6
inserting into database 6-6
multibyte characters 3-9, 3-36
nonprintable characters 3-10

O

OCTET_LENGTH function 3-29
onaudit utility 4-4
oncheck utility 4-4
ONCONFIG configuration parameters 1-1
onload utility 4-4
onlog utility 4-4
onmode utility 1-6
onpload utility 4-4
onshowaudit utility 4-4
onspaces utility 4-4
onstat utility 4-4
onunload utility 4-4
Opaque data types 3-2, 3-33, 4-8, 4-16
identifier 3-2
Operating system
8-bit clean 1-9, 2-8
character encoding 1-30
limitations 6-3
need for code-set conversion 1-30
saving disk space A-11
Operator class 3-2
ORDER BY clause (SELECT) 1-11, 3-19
ORDER SIBLINGS BY clause of SELECT statements 3-16
Owner name 3-3

P

Parameter marker 4-13
Partial characters 3-15, 5-7
path name 3-2
Path name 3-4
Percent (%) symbol 4-13
PREPARE statement 3-2
Pseudo-user 3-3

Q

Question (?) mark wildcard 3-26
Quoted string 3-4, 3-12

R

Range matches 3-24
rdatestr() library function 1-13, 6-7
rdefmtdate() library function 6-7, 6-8
rdownshift() library function 6-14
receive() function 4-17
registry file 1-9, A-10, A-11
Regular expression 1-6, 1-19

Relational-operator conditions 3-21
RENAME COLUMN statement 3-1
Resource file 5-5
rfmtdate() library function 6-7, 6-8
rfmtdec() library function 6-11
rfmtdouble() library function 6-11
rfmtlong() library function 6-11
rgetlmsg() library function 5-5
rgetmsg() library function 5-5
Right-to-left writing direction 1-19
Role 3-2
Round-trip conversion 1-29
ROW data types 3-2
rstrdate() library function 6-7
Runtime error, custom message 4-10
rupshift() library function 6-14

S

Schema name 3-3
Screen reader
  reading syntax diagrams B-1
Search functions 3-18
SELECT statements
  and collation order 1-11
  collation of character data 3-18, 3-19
  embedded 3-34
  LIKE keyword 3-26
  MATCHES relational operator 3-24
  ORDER BY clause 1-11, 3-19
  select-list columns 6-16
  specifying literal matches 3-24, 3-26
  specifying matches with a range 3-24
  specifying quoted strings 3-12
  using length functions 3-27
  using TRIM 3-18, 6-17
  WHERE clause 1-11, 3-21
send() function 4-17
Sequence 3-2
Server code set 1-29
Server computer
  server code set 1-29
  setting DB_LOCALE 1-23
  setting SERVER_LOCALE 1-23
Server locale
  code set 1-29
  definition of 1-21
  in trace messages 4-13
  setting 1-23
  uses of 4-1
SERVER_LOCALE environment variable 1-3, 2-21
  database server filenames 4-1
  default value 1-23
  example of locale name 2-21
  location of message files 2-4
  precedence of 2-4
  role in code-set conversion 4-2
  setting 1-23
  syntax 2-21
  with utilities 4-3
Server-processing locale
  code-set conversion 4-2
  COLLATION category 1-27
  CTYPE category 1-27
  date data 1-28
  definition of 1-26
  determining 1-26
Server-processing locale *(continued)*
  filename checking 4-2
  for exception messages 4-12
  initialization of 1-26
  localized collation 1-27
  MESSAGES category 1-28
  MONETARY category 1-28
  monetary data 1-28
  NUMERIC category 1-28
  numeric data 1-28
  precedence of environment variables 1-27, 1-28
  TIME category 1-28
  time data 1-28
  UDRs and 4-5
SET clause of INSERT or MERGE 3-34
SET COLLATION statement 1-12, 2-5, 3-18, 3-24, A-4
SET EXPLAIN statement 1-21
SET NO COLLATION statement 1-12
Shortcut keys
  keyboard B-1
Single quotes 3-4
Single-byte characters 1-9, 3-14, 3-16
SMALLFLOAT data type A-4
SMALLINT data type A-4
Spanish locale 1-24
SPL routines 1-6, 3-2
SQL API products
  comments 6-1
  ESQL library enhancements 6-7
  filenames 6-1
  host variables 6-1
  literal strings 6-1
  SQL identifier names 6-1
  using GLS8BITFSYS 2-7
SQL functions for case 3-18
SQL identifier
  delimited 3-2
  examples 3-5
  non-ASCII characters 4-6, 6-1
  owner names 3-3
  rules for 3-1
SQL length function
  CHAR_LENGTH 3-30
  classification of 3-27
  LENGTH 3-27
  OCTET_LENGTH 3-29
  using 3-27
SQL segments 3-3
SQL statements
  ALTER TABLE 3-36
  CONNECT 3-2
  CREATE CAST 3-2
  CREATE DISTINCT TYPE 3-2
  CREATE FUNCTION 3-2
  CREATE INDEX 3-1, 3-2, 3-18
  CREATE OPAQUE TYPE 3-2
  CREATE OPCLASS 3-2
  CREATE PROCEDURE 3-2
  CREATE ROLE 3-2
  CREATE ROW TYPE 3-2
  CREATE SEQUENCE 3-2
  CREATE SYNONYM 3-2
  CREATE TABLE 3-36
  CREATE TRIGGER 3-2
  CREATE VIEW 3-2
  data definition 3-36
  data manipulation 3-34

SQL statements *(continued)*
DECLARE 3-2
DELETE 3-34
DESCRIBE 6-16
end-user formats in 1-13
FETCH 3-2
in code-set conversion 4-6, 5-3
in UDRs 4-6
LOAD 3-2, 3-34, 3-35
MERGE 3-34
PREPARE 3-2
RENAME COLUMN 3-1
SELECT 3-16
SET COLLATION 3-18, 3-24, A-4
SET COLLATION statement 1-12
SET EXPLAIN 1-21
UNLOAD 3-34, 3-36
UPDATE 3-34
SQL utilities 4-4
SQL_LOGICAL_CHAR configuration parameter 3-36
SQL-99 standard 3-13
SQLBYTES data type constant 6-15
sqlca structure
connection warnings 1-25
sqlerrm 5-3
SQLWARN array 1-25, 1-26, 1-27, 5-1
sqlwarn.sqlwarn7 1-26
warning character 1-25
sqlca.sqlwarn.sqlwarn7 flag 1-26
sqlda structure 6-14, 6-16
sqlda.sqlvar.sqldata field 6-16
sqlda.sqlvar.sqllen field 6-16
sqlda.sqlvar.sqlname field 6-17
SQLSTATE status value 4-10
SQLTEXT data type constant 6-15
sqltypes.h header file 6-15, 6-17
sqlvar_struct structure
description of 6-16
sqldata field 6-16
sqllen field 6-16
sqlname field 6-17
storing column data 6-16
SQLWARN warning flag 1-25, 1-26, 1-27, 5-1
standards x
Statement identifier 1-6, 3-2, 4-6, 5-3, 6-1
Substitution conversion 1-29
Substring 3-13, 3-17
Synonym 3-2
Syntax diagrams
reading in a screen reader B-1
syserrors system catalog table 4-10, 4-12
systables system catalog table 1-19
System catalogs 1-19
systracemsgs system catalog table 4-13, 4-14, 4-15

T

Table (database)
in code-set conversion 5-3
naming 1-6, 3-2, 4-6, 6-1
Taiwanese dates 1-15, 1-33
Territory 1-22, 1-25, 2-2, 2-5, 2-21, A-6
TEXT data type
code-set conversion 5-3
collation order 1-13
GLS aspects 3-12
in code-set conversion 6-14
TEXT data type *(continued)*
partial characters 3-17
Thousands separator 1-15, 1-22, 3-32, 6-11, A-4, A-5
Time data
customizing format of 1-32
end-user format 1-22, 1-28, 1-32, A-5
format of A-5
locale-specific 1-6, 1-13
precedence of environment variables 1-33, 6-10
with DBTIME 2-6
with GL_DATETIME 2-15
TIME locale category
description of A-2, A-5
end-user formats A-5
era information 2-12, 2-17, A-5
in client locale 1-18
in locale source file A-6
in server-processing locale 1-28
Token names 4-13
Top-to-bottom writing direction 1-19
Trace block 4-14
Trace message 4-13
Tracing
GL_DPRINTF macro 4-14
gl_tprintf() function 4-14
trace blocks 4-14
trace message 4-15
Triggers 3-2
TRIM function 3-12, 3-18, 6-17

U

Unicode
Basic Multilingual Plane (BMP) 1-30
Collation Algorithm 1-13
dbspaces 1-13
index keys 1-13
nondefault page size 1-13
UTF-8, UTF-16, and UTF-32 code sets 1-9
Unified Chinese code set (GB18030) 1-9, 1-30
UNIX environment
default locale 1-22
glfiles utility 2-2, 2-5, 2-21
supported code-set conversions 5-1
supported locales 2-2, 2-5, 2-21
Unload file 3-36
UNLOAD statement 3-34, 3-36
Unsigned short 4-7
UPDATE statements
embedded SELECT 3-34
era-based dates 3-34
GLS considerations 3-34
SET clause 3-34
WHERE clause conditions 3-34
UPPER function 3-12, 3-18
USE_DTENV environment variable 2-20
User-defined function 3-2
User-defined procedure 3-2
User-defined routine (UDR)
character strings in 4-7, 4-8
code-set conversion in 4-8
current processing locale 4-5
exception messages 4-10
file names 4-6
Globalized 4-5
IBM Informix GLS API 4-7
literal strings 4-6

User-defined routine (UDR) *(continued)*
locale support 4-5
non-ASCII source code 4-5
SQL identifier names 4-6
trace messages 4-13
User-defined routines 3-2
UTF-8 character encoding 2-20
UTF-8 code set 1-9
Utilities 1-6
chkenv 4-4
database server 1-6
database server utilities 4-3
DB-Access 1-6, 4-4
dbexport 1-6, 4-4
dbimport 4-4
dbload 4-4
dbschema 4-4
glfiles 2-2, 2-5, 2-21, 5-1, A-12
onaudit 4-4
oncheck 4-4
onload 4-4
onlog 4-4
onmode 1-6
onpload 4-4
onshowaudit 4-4
onspaces 4-4
onstat utility 4-4
onunload 4-4
SQL utilities 4-4
supporting multibyte characters 4-3

V

VALUES clause of INSERT or MERGE 3-34
VARCHAR data type
and GLS 1-7
code-set conversion 5-3
collation order 1-13
difference from NVARCHAR 3-9
GLS aspects 3-11
multibyte characters 3-36
View 1-6, 3-2, 4-6, 6-1
Visual disabilities
reading syntax diagrams B-1

W

W warning character 1-25
Warnings 1-25, 1-26, 1-27, 5-1
custom 4-10
wchar_t data type 4-7
WHERE clause
and collation order 1-11
BETWEEN condition 3-22
IN condition 3-23
in DELETE statement 3-34
in INSERT statement 3-34
in MERGE statement 3-34
in UNLOAD statement 3-34
in UPDATE statement 3-34
logical predicates 3-21
relational-operator condition 3-21
White space
in formatting directives 2-10, 2-15
in locale A-3
Wide character 4-7
Wildcard character 3-26
Windows environments
default locale 1-22
supported code-set conversions 5-1
Writing direction 1-19

Y

Year 0000 1-15

Z

Zeros in number values 3-7

Printed in USA

SC27-4499-01

Spine information:

Informix Product Family Informix Global Language Support **Version 6.00**

IBM Informix GLS User's Guide